Solar Flare: Solar Plant Data Acquisition & Visualization

Felipe Velosa, Vincent Giannone

Final Draft
12/5/2013

CNT 4104 Software Project in Computer Networks
Instructor: Dr. Janusz Zalewski
Computer Science & Software Engineering Programs
Florida Gulf Coast University
Ft. Myers, FL 33965
1. Introduction

The purpose of this document is to describe our implementation of the analysis and visualization of the data retrieved from the 16-acre solar farm on the Florida Gulf Coast University campus, specifically the solar panels powering Academic Building 7. This project improves upon other projects previously researched, such as Solar Cloud - Rapid Data Logging from a Solar Power Plant [1], daVinci: Solarems Data Extraction [2], and daVinci: eBox 4864 – Sentalis Fetch CSV Server [3]. These papers discuss the process of retrieving, storing and accessing the data acquired from the solar panels. Figure 1 illustrates the interconnectivity between the Solar Farm, AB7 Panels (3 PADS), and the data-logging unit (Sentalis 1000).

Figure 1. Interconnectivity between Solar Farm, AB7, and data logging unit
The previous projects were highly dependent on browser automation, which require high maintenance. Upon further research we found a more effective way of retrieving data from the Solar Farm. We are proposing the use of HTML GET and POST request to access the data retrieved from the Solar Farm via the solarems.net website. Instead of browser automation we want to emulate a browser programmatically using java.net. Browser emulation and data acquisition will be delegated to a software named SolarFlare. SolarFlare will also feed data acquired into a database included in the server program package. These data will be read and visualized in an HTML file named index.html. This HTML file is also included in the server program package. Figure 2 represent the server program package and the internal flow of data.

![Diagram](image)

Figure 2. Server Program Package and flow of data

We are using the Highcharts interactive Javascript charts API to visually represent acquired data. Highcharts is an interactive client side scripting tool that is embedded into a script tag. Our implementation of Highcharts uses PHP to read from the database in the SolarFlare software package. Figure 3 is a screenshot of the data displayed on solarems.net. SolarFlare acquires the data by searching for keywords and tags in the HTML source code of
solarems.net. These data are extracted and formatted into a SQL query so it can be fed into the server program database.

![Table of Solar Panel Data]

**Figure 3. Screenshot of the data on solarems.net**

The Highcharts code embedded in the HTML page queries the database to generate the graphs. These graphs are presented in an interface that will allow a browser to view the data by day, week or year. This design will allow anyone to view the data acquired from the solar plant via a browser. Figure 4 and Figure 5 are examples of charts generated by Highcharts. These charts plot wind speed, temperature, and horizontal irradiance of an array of solar panels over a period of time.
To complement these charts, additional explanation is needed on the technical terms that describe the status of the Solar Farm technology, which will be accomplished by describing the data being visualized. Figure 6 shows a prototype of a temperature vs. time chart where the analysis is below the chart. This analysis provides additional information about the temperature. The website will also include descriptions of terms such as: Horizontal Irradiance, Weighted Average PO Irradiance, DVC current, AC current, Frequency, Kilowatts, etc.
Figure 6. Prototype of Temperature vs. Time including terminology description

The graph above plots temperature vs. time. **Temperature**: the degree or intensity of heat present in a substance or object, esp. as expressed according to a comparative scale and shown by a thermometer or perceived by touch.

The next section presents the definition of the problem and outlines the technical details of the project.
2. Definition of the Problem

The purpose of this project is to acquire, visualize and analyze the data from the 16-acre solar farm that powers Academic Building 7 on the FGCU campus. The original projects, which included Solar Cloud - Rapid Data Logging from a Solar Power Plant [1], daVinci: Solarems Data Extraction [2], and daVinci: eBox 4864 – Sentalis Fetch CSV Server [3], acquire and insert data into a database. This database is not being used in the current project. Furthermore the way data are acquired in previous projects is unreliable because the software that acquires data is only an automation of a browser, whose execution can be inadvertently affected by a keystroke or mouse movement. Considering visualization relies on constant real-time data acquisition, this way of acquiring data from solarems.net does not meet common standards of reliability. Since the automation process relies on filling the visual forms in the HTML source code of solarems.net, every time a form changes, the existing project crashes because automation relies on the graphical representation of these forms. HTML GET and POST requests directly communicate with the web server to send information and retrieve HTML source code. These requests don’t rely on external web browsers or graphical representations of web forms.

This project improves on the data acquisition system by programmatically connecting, verifying credentials, navigating, and acquiring data. Figure 7 shows an example of code of POST request written in Java, which emulates a browser, stores cookies, connects to solarems.net, and submits credentials via POST request. This implementation is more reliable than the previous one because it does not depend on the automation of a web browser.

```java
private void sendPost(String url, String postParams, String refer) throws Exception {
    URL obj = new URL(url);
    conn = (HttpURLConnection) obj.openConnection();

    // Acts like a browser
    conn.setUseCaches(false);
    conn.setRequestMethod("POST");
    conn.setRequestProperty("Host", "solarems.net");
    conn.setRequestProperty("User-Agent", "USER_AGENT");
    conn.setRequestProperty("Accept", "text/html,application/xhtml+xml,*/*");
    conn.setRequestProperty("Accept-Language", "en-US");

    for (String cookie : this.cookies2) {
        conn.setRequestProperty("Cookie", cookie.split(";", 1)[0]);
    }
    conn.setRequestProperty("Connection", "keep-alive");
    conn.setRequestProperty("Referer", refer);
    conn.setRequestProperty("Content-Type", "application/x-www-form-urlencoded");
    conn.setRequestProperty("Content-Length", Integer.toString(postParams.length()));

    conn.setDoOutput(true);
    conn.setDoInput(true);
}
```

Figure 7. POST Request and browser emulation
The solar farm that powers Academic Building 7 is composed of three solar panel arrays (Pads). These arrays are Pad A, Pad B1, and Pad B2 generating a maximum amount of energy at 539.46 kW, 269.73 kW, and 263.07 kW, respectively. This project uses the Draker’s Sentalis 1000 Photovoltaic (PV) data monitoring system, which acquires data at fifteen minute intervals from the farm. These data are accessible from the solarems.net website, which is a web application that interfaces with the data logger and allows a user to generate and export a data set. The data available on the site include individual and global attributes for all three Pads: environmental values, a load meter for Academic Building 7, Plane of Array (POA) irradiance, horizontal irradiance, and PV cell temperature. The purpose of this project is to analyze and visualize these data in a way that is educational to the viewer. Students, faculty and the community could view the graphs and learn what the solar farm is producing.

This project is implemented using HTML, Javascript, Java and PHP. This implementation is broken down into four phases.

- First we will programmatically login and navigate the solarems.net website using CONNECT, POST, and GET requests.
- Second an application will parse the data results page looking for <span> tags in the HTML code that represents the values to be graphed. An example of these tags is shown in Figure 8.
- Third, each value will be recorded into a database using SQL commands.
- Fourth, Highcharts will interface with this database dynamically showing the data as they are retrieved from solarems.net in 15 minute intervals.
Figure 9 shows the context diagram for the project’s system and its components. There are three entities that interact with each other using HTTP requests: solarems.net, the server application and the client. The server in the lab will store SolarFlare, a SQL database, and the webpage that implements the Highcharts API. SolarFlare, a server application written in Java, sends HTTP POST requests to solarems.net to login. HTTP GET requests are also sent by SolarFlare to retrieve data from solarems.net. These data are recorded into a database located within the application package. The database generates a unique identification number for each successful GET request from solarems.net. When a client accesses the web page by opening index.html, Highcharts will traverse the database and graph the data. Highcharts runs on the client side, where PHP will run and dynamically update the graph. Each graph will have its own analysis that explains what it represents. This way the viewer becomes educated on what is being acquired from the solar farm.
Figure 10 shows the code sample how the data will be retrieved from the database and formatted for Highcharts to display in the webpage. Since this is an example, PHP sends a query to the database asking for every field and all records in those fields to be retrieved. The query returns these data and separates them by the x-axis and y-axis.
//execute the SQL query and return record
$queryReturn = mysql_query("SELECT * FROM " . $db_table);

// Fetch columns in desired axis
while($row = mysql_fetch_array($queryReturn)) {
    $x_axis[] = $row['id'];
    $y_axis[] = $row['temp'];
}

// Create output string in for [[x1, y1], [x2, y2], ..., [xn, yn]]
$x_axis = NULL;
if($y_axis && $x_axis) {
    $points_string = "["
    $i = 0;
    while ($i < sizeof($y_axis)) {
        if($i == sizeof($y_axis))
            $points_string = $points_string . $x_axis[$i] . "," . $y_axis[$i] . "]";
        else
            $points_string = $points_string . $x_axis[$i] . "," . $y_axis[$i] . "]", [";
        $i = $i + 1;
    }
}

Figure 10. PHP code that creates an output string for Highcharts to use as data

The next section describes the design solution and implementation of the proposed application. It continues the top-down perspective to illustrate the operation of the software.
3. Design Solution

The purpose of this project is to develop software that acquires and visualizes data that describes the status of the Solar Farm technology that powers Academic Building 7 (Figure 11) and educates a user on the technology used. Section 2 defined the problem that the respective application will solve and outlined its essential components. This section describes the elements that interact in the system and how they contribute to meeting the objectives of section 2. It essentially addresses the question: How can data be retrieved from the solar farm and visualized in an educational matter?

![Figure 11. Academic Building 7](image)

Figure 12 displays the five physical components of the entire system. These components include the Solar Farm, Sentalis data logger, the solarems.net website, the client, and the SolarFlare with an Internet connection. Figure 12 shows these components, and their interactions with SolarFlare as the central entity with whom all communication and functionality is controlled.

The SolarFlare Application initially communicates with the Solar Farm via the solarems.net to access Sentalis, the data acquisition unit. Sentalis provides data to SolarFlare via solarems.net, which is a dynamic web based data provider.
solarems.net updates its text tags in 15 minute intervals with real-time data from Sentalis. A screenshot of these data is shown in Figure 3. SolarFlare sends HTTP GET and POST requests to solarems.net to login programmatically and retrieve the web pages that show these data. The solarems.net server sends the HTML files to the SolarFlare Server via the HTTP GET request. All of this communication follows the Open Systems Interconnection (OSI) model via the Internet using the Transmission Control Protocol (TCP) and Internet Protocol (IP). With all these components interacting in unison, SolarFlare responds to user HTTP Get requests to obtain HTML pages, that include the visualization and analysis of the Solar Farm status.

Figure 12. Physical Diagram

Using these components and their interconnections, the flow of operations in the SolarFlare Application can be described as shown in Figure 13.1. It illustrates the Application Flowchart of SolarFlare. Initially, the Application starts on the server where the Java Virtual Machine (JVM) is initialized and the program begins execution. The application then connects to
solarems.net. After connecting to solarems.net, it must verify credentials by sending HTTP POST requests to the solarems.net server. Once the credentials are verified, the server will allow SolarFlare access to it. SolarFlare will then send an HTTP GET Request, asking for the HTML page that has the data available on it. SolarFlare will then parse the page with the data. These parsed data are fed to a database available on the SolarFlare Server. After each generation SolarFlare will sleep for 15 minutes. The sleep is required since solarems.net is updated in 15 minute intervals. This action will protect our software from acquiring redundant data. This action will not stop a client from requesting the index.html at anytime. index.html refers to the index file of the website that charts all of the data stored in the database by SolarFlare.

Figure 13.2 illustrates the Application Flowchart for the website that charts the data points contained in the database. This website uses php code to query the database. This query happens everytime index.html is accessed. The data are retrieved and stored as a JSON file, which is then read by the Highcharts API to generate the charts.
Figure 14 shows the class diagram for SolarFlare, which is implemented as a sequence of modules. These modules are represented as the classes `SolarFlare`, `HTMLNavigator`, `HTMLParser`, and `DBManager`. Each class has functions and attributes that will support the logic of the software. `HTMLNavigator` is a class that is used to send HTTP POST and GET requests to the `solarems.net` website to login and navigate to the webpage that has these data available on it. The `cookies`, `conn`, `USER_AGENT`, and `login` attributes are used, along with the `login()`, `sendPost()`, `GetPageContent()`, `Fetch()`, and `GetParamForms()` functions, to accomplish this task. `HTMLParser` is a class that parses the webpage, represented as a String returned from `HTMLNavigator`, that encapsulates the data to be uploaded to the database. The `scan` attribute is used, along with the `getValue()` function, to accomplish this task. The `DBManager` class establishes a connection with the database, enabling queries to be executed. The `conn`, `stmt`, `rs`, and `DatabaseName` attributes, which are used by the `Connect()`, `Disconnect()`, `Update()`, and `Query()` methods, allow for this task to be accomplished. The `SolarFlare` class is the main module of the software and is
a Thread that executes the logic in run(), that uses these classes to accomplish the acquisition and visualization of data from the solarems.net website.

Figure 14. Class Diagram

The next section, describes the implementation of SolarFlare. It continues the top-down perspective to illustrate the functionality of the SolarFlare, HTMLNavigator, HTMLParser, and DBManager classes in code.
4. Implementation

SolarFlare is implemented in a modular fashion via the Java programming language. The driving class, SolarFlare, implements the Runnable interface, allowing the software to be executed as a Thread class that executes these modules. The modules, which are represented in respective classes, are SolarFlare, HTMLNavigator, HTMLParser, and DBManager, shown in Appendix A. The main method creates a Thread object.

```java
Thread t = new Thread(new SolarFlare());
```

The method start() is then called to place the thread in the running state, beginning execution at the run() method in SolarFlare. In run(), all logic is in the scope of the while loop which runs infinitely so that it will start on the server and execute autonomously. The logic begins by declaring and initializing two objects, HTMLNavigator and DBManager. HTMLNavigator will navigate the solarems.net website by logging in and accessing the data webpage. The DBManager object establishes a connection with the database and enables queries to be sent. Next, the login() method is called by the HTMLNavigator object, logging into the website. username and password are strings that are sent via an HTTP POST request to solarems.net to login. Afterwards, the urls specified in urlsToNavigate are fetched in sequence using the fetch() method of the HTMLNavigator object and returned as an HTMLPage. This page, which is where the data are located, is then parsed by the HTMLParser object. The parsing process is in the scope of a while loop that iterates while attributes are still available on the page https://solarems.net/projects/36-fgcu-ab7/data_sets/26.

Figure 3 in section 1 shows how these data are formatted on that page. After the data have been parsed from the page and stored as a string, an SQL statement is prepared and sent via the update() method available in DBManager to the database. The thread then sleeps for 16 minutes via a call to the Thread.sleep(960000) method. This is done because the data page on solarems.net is updated in 15 minute intervals. Once it has slept, the while loop iterates again, repeating the logic. A discussion of the HTMLNavigator, HTMLParser, and DBManager classes follow, along with important segments code that describe their core logic.
HTMLNavigator was developed out of the necessity to login in solarems.net and fetch the web page where the data is located. Figure 15 and Figure 16 show the login() and fetch() methods.

```java
public boolean login(String loginUrl, String referenceUrl, String username, String password) {

    // make sure cookies are turned on
    CookieHandler.setDefault(new CookieManager());

    try {
        // REQUIRED: NAVIGATING SITE AQUIRES COOKIES. THIS IS NEEDED BEFORE POST
        GetPageContent("https://www.solarems.net");

        // 1. Send a "GET" request, so that you can extract the form's data.
        String postParams = getFormParams(username, password);

        // 2. send a POST request for authentication
        System.out.println("AUTHENTICATING....");
        sendPost(loginUrl, postParams, referenceUrl);
        System.out.println("GREAT SUCCESS!\n");

        // the class has logged in sucessfully
        login = true;
        return true;
    }

    catch (Exception e) {
        // Class did not login
        login = false;
        return false;
    }
}
```

**Figure 15. login() method in HTMLNavigator**

login() takes the loginUrl, referenceUrl, username, and password strings as parameters. First, the CookieHandler object calls on the setDefault() method to enable cookies on the web page accessed. Then a get request is sent to solarems.net with the username and password passed as strings via getFormParams().getFormParams() sets the appropriate attributes to of the HTML form. This form is returned to the postParams string, which is then passed to the sendPost() method. sendPost() connects to solarems.net and write the bytes of
postParams to an outputstream to the solarems.net server. If no exception is thrown, then the login process was successful, setting the login class attribute to true.

The fetch() method shown in Figure 16 retrieves the HTML content of the web page specified in the urls string passed as a parameter. The urls that are retrieved were declared in the SolarFlare code in Appendix A, and they are https://solarems.net/projects/36-fgcu-ab7/overview and https://solarems.net/projects/36-fgcu-ab7/data_sets/26, represented as url[0] and url[1], respectively. In the first iteration of the for loop, url[0] is overwritten. It is not necessary for the software to save the content retrieved from this page. But, the content retrieved from url[1] is saved and assigned to result. result is returned back to the SolarFlare module for parsing by the getValue() method.

```java
public String fetch(String[] urls) throws Exception {
    String result = null;

    // Make sure the user has logged in
    if (login) {

        // for every link in the url array
        for (int i = 0; i < urls.length; i++) {
            System.out.println("RETRIVING: " + urls[i]);
            // fetch the page content
            result = getPageContent(urls[i]);
            System.out.println(result);
        }

        // return the last website fetched
        return result;
    } else {

        System.out.println("Please login first.");
        return null;
    }
}
```

Figure 16. fetch() method in HTMLNavigator
The `getValue()` method in Figure 17 takes the string `attributeName` as a parameter. `attributeName` is a string that defines the type of data that are sought by the caller. These attributes are retrieved from a file indicated in the `SolarFlare` class in Appendix A. This file can be changed at anytime by anyone who is maintaining the software. It is a place where variables that have been added or taken away from `solarems.net` can be accurately represented. This is how the software is told which attributes to read from the web content.

Suppose the caller wanted to get the value of Horizontal Irradiance. It would be passed to `getValue()` as `attributeName`. The method begins execution by first establishing a delimiter. This string is used as a way of telling the scanner to only read values after that string has appeared in its datastream. Since there may be many occurrences of this delimiter, and there are, a while loop is used to iterate through the stream until the string returned to `currentLine` by the `next()` method is equal to `attributeName`; once it is, the data to be read by has been found. Now, a new delimiter is set to bring the scanner to the numerical value in the stream; assuming a standard has been followed of placing the data after the delimiter chosen. All numbers following the delimiter are concatenated to a string until white space is reached. This logic is represented in the embedded for loop. The value is stored in the `strVal` string.

Once the numerical value has been read from the datastream via scanner, it will be converted to a double by the `parseDouble()` method (shown in Figure 18). If there was no error in reading the value of the attribute specified from the HTML content, the string will be parsed and passed back to the caller with no error. If there was an error in reading, or the value was not available, or the attribute does not exist on the page, -9999 error code will be returned to the caller.
public double getValue(String attributeName) {
    Scanner byName = scan.useDelimiter("name"">");

    while (byName.hasNext()) {
        String currentLine = byName.next();

        if (currentLine.contains(attributeName)) {
            Scanner byValues = new Scanner(currentLine);
            byValues = byValues.useDelimiter("value"">");

            byValues.next();

            if (byValues.hasNext()) {
                String valueString = byValues.next();

                char[] c = valueString.toCharArray();

                String strVal = "";
                for (int i = 0; i < c.length; i++) {
                    if (c[i] == '0' || c[i] == '1' || c[i] == '2'
                        || c[i] == '3' || c[i] == '4' || c[i] == '5'
                        || c[i] == '6' || c[i] == '7' || c[i] == '8'
                        || c[i] == '9' || c[i] == '.' || c[i] == ','
                        || c[i] == '-' ) {
                        if(c[i] == ','){
                            continue;
                        }
                        strVal = strVal + c[i];
                }
            }
            try {
                double value = Double.parseDouble(strVal);
                return value;
            } catch (Exception e) {
                return -9999;
            }
        }
    }
}

Figure 17. getValue() method in HTMLParser

Figure 18. Segment from getValue() method in HTMLParser; return value
After all values of the specified attributes have been retrieved, the database will be updated with these values via a call to the `update()` method in the `DBManager` class. The logic of the `update()` method is shown in Figure 19. A string, `SQL_Update`, is passed as a parameter to the method which represents the query to be sent to the database. This query will create a new record in the database with the values retrieved from `solarems.net`.

```
public boolean update(String SQL_Update) {
    try {
        stmt = conn.createStatement();
        stmt.setQueryTimeout(30);

        try {
            // execute update
            stmt.executeUpdate(SQL_Update);
            stmt.close(); // close statement
            conn.close(); // close connection
        }
        catch (Exception e) {
            System.out.println("WARNING: Record was not updated.");
            System.out.println("Update Query: " + SQL_Update);
            return false;
        }
    }
    catch (Exception e) {
        System.out.println("WARNING: Failed to create statement");
        return false;
    }
    return true;
}
```

**Figure 19. update() method in DBManager**

First, the `Statement` object, `stmt`, is initialized the `createStatement()` method call from the `Connection`, `conn`, object. The database is then updated by the statement, via the connection, with the arguments and fields specified in `SQL_UPDATE`. If an error occurs when updating, an exception is thrown. The error is reported to the user.
The code for the SolarFlare and HTMLParser classes has been added to Appendix A to support the segments of code represented in Figures 15 through 19.
5. Experimentation

SolarFlare is deployed as a Java application that runs on a server. This application has an execution thread that runs every 15 minutes. The application runs on the background and has three main goals:

1. Log into solarEMS.net programmatically and navigate towards the data section.
2. Crawl this section to extract data from its HTML code
3. Store the extracted data into a database that can be reached remotely.

Figure 20 shows the root folder of SolarFlare. This directory contains the executable JAR file for SolarFlare (SolarFlare.jar), the SQLite database (SolarFlareDB.sqlite), an Attributes file containing the same names of the attributes in the SolarEMS page to extract and a library folder containing the SQLite driver (sqlite-jdbc-3.7.2.jar).
To test goals 1-3 SolarFlare was executed and the SQLite database was monitored for changes. To run SolarFlare open the SolarFlare.jar file in the root directory. Figure 21 shows the SQLite database with one record in its database and SolarFlare getting ready to run. Figure 22 shows the database with two additional records after 15 minutes of running SolarFlare.

Figure 21. SolarFlare beginning of execution
Figure 22. SolarFlare after 15 minutes of execution

Figure 23 displays the console output by SolarFlare after successful execution of updating the database. There are three distinct parts to the output. The first is the authentication process, which is the process of logging into solarEMS.net. After the authenticating text, a post request is sent to https://solarems.net/user_session and sends a response code of 200 upon success. This second part is that the web page that displays the data is retrieved https://solarems.net/projects/36-fgcu-ab7/data_sets/26. The next and final part is the acquisition of the data associated with each attribute displayed from the webpage.

Figure 24 displays the solarEMS.net the webpage https://solarems.net/projects/36-fgcu-ab7/data_sets/26 and the attributes in the Attribute.txt file that SolarFlare uses to extract from that HTML code. This follows the parts shown in Figure 23, where the software successfully submits a post
request to solarEMS.net, extracting those data values based on the attribute field names, placing them successfully inside the SQLite database.

```
RUN:
AUTHENTICATING....
Sending 'POST' request to URL : https://solarems.net/user_sessions
Response Code : 200
GREAT SUCCESS!

Target: https://solarems.net/projects/36-fgcu-ab7/data_sets/26
RETRIVING: https://solarems.net/projects/36-fgcu-ab7/overview
RETRIVING: https://solarems.net/projects/36-fgcu-ab7/data_sets/26
Target HTML Retrived.

Aquiring Attributes...

Aggregate PV Index [Base]: 0.0
Aggregate PV Index [Derated]: 0.0
Weighted Avg POA Irradiance: 0.54
Aggregate AC Power: -0.03
Aggregate Modeled AC Power [Base]: 0.0
Aggregate Modeled AC Power [Derated]: 0.0
Imported Power with PV: 173.06
Imported Power without PV: 173.03
Aggregate AC Energy Record: 0.0
Aggregate AC Energy Total: 5312627.0
Total Delivered Solar Energy From Load: 0.0
AC Power Factor: -0.91
COMM Status: 0.0
AC Current Neutral: 29.83
AC Current Phase A: 232.4
AC Current Phase B: 194.0
AC Current Phase C: 205.5
AC Frequency: 59.96
AC Voltage AN-AB7: 288.0
AC Voltage BN-AB7: 289.0
AC Voltage CN-AB7: 289.1
AC Apparent Power: 186205.5
AC Power: -0.03
AC Energy Load Record: 38.0
AC Energy Total: 5312627.0
DataLogger Temperature: 29.15
DataLogger Voltage: 12.85
POA Irradiance: 0.54
Cell Temperature: 24.74
Horiz Irradiance: 0.0
Ambient Temperature: 0.0
Wind Speed: 3.31
Wind Speed Max: 6.5
Wind Direction: 42.8
Connecting to database:
Database updated successfully
```

Figure 23. SolarFlare console output
The webpage developed that visualizes the data retrieved from the SolarFlare acquisition software is called FGCUSolar and shown in Figure 25. It is written in HTML 5 and uses Cascading Style Sheets (CSS), PHP: Hypertext Preprocessor (PHP), and JavaScript to query the SQLite database updated by SolarFlare. This website application has three main goals:

1. Retrieve data from the database
2. Encode JSON files
3. Populate and display a Highchart graph of data read from JSON file

When these goals are successful they output the final product shown in Figure 25. There are three pages that graph eleven sets of data. The page shown below graphs the equipment data sets, datalogger temperature and cell temperature.
These goals were tested by generating the graph shown in Figure 25, Datalogger Temperature vs. Time, which was successful. First the data to be graphed was defined in Figure 26. $attribute1 was set to “Datalogger Temperature”. Then a JSON file is generated by calling $getData() and passing to it the attribute to be retrieved from the database. $getData() returns the relative path of this file. The code in Figure 27 shows how the data retrieved and the JSON files are encoded to satisfy goals 1 and 2.

```
// Define data to be charted
$attribute1 = "Datalogger Temperature";
// Generate JSON file
$file1 = $getData($attribute1);
```

Figure 26. Define graph to generate and get data
**Figure 27. Code to generate JSON file**

The JSON file was generated successfully and is shown below in Figure 28 and its contents are in Figure 29. The filename uniquely identifies each user that accesses the website by the `session_id()` and the `getTimestamp()` function calls.
Figure 28. JSON file in directory

<table>
<thead>
<tr>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>2i.hlgn04bdvt9fqf4ujb8rgiq2Datalogger Temperature1383882513.json</td>
</tr>
<tr>
<td>2i.hlgn04bdvt9fqf4ujb8rgiq2Horizontal Irradiance1383882510.json</td>
</tr>
<tr>
<td>2i.hlgn04bdvt9fqf4ujb8rgiq2POA Irradiance1383882511.json</td>
</tr>
<tr>
<td>bcsikfhep5vm405gahan8r21u4Datalogger Temperature1383882481.json</td>
</tr>
<tr>
<td>bcsikfhep5vm405gahan8r21u4Horizontal Irradiance1383882477.json</td>
</tr>
<tr>
<td>bcsikfhep5vm405gahan8r21u4POA Irradiance1383882479.json</td>
</tr>
<tr>
<td>fmprbk2l1d7sfe4jdbibhqcn0Datalogger Temperature1383882428.json</td>
</tr>
<tr>
<td>fmprbk2l1d7sfe4jdbibhqcn0Horizontal Irradiance1383882424.json</td>
</tr>
<tr>
<td>fmprbk2l1d7sfe4jdbibhqcn0POA Irradiance1383882426.json</td>
</tr>
</tbody>
</table>

Figure 29. Contents of JSON file for Datalogger Temperature points

```
[1270140300000, 52.16],
[1270141200000, 51.45],
[1270142100000, 52.38],
[1270143000000, 53.4],
[1270143900000, 52.97],
[1270144800000, 52.41],
[1270145700000, 52.45],
[1270146600000, 53.11],
[1270147500000, 53.23],
[1270148400000, 48.27],
[1270149300000, 47.2],
[1270150200000, 42.24],
[1270151100000, 38.47],
[1270152000000, 34.78],
```

The Javascript code shown in Figure 30 uses the Highstock API for graphing the chart in Figure 25. This code has been tested and works, also satisfying goal three.
Figure 30. Code to generate Highcharts

```javascript
// Apply the theme
var highchartsOptions = Highcharts.setOptions(Highcharts.theme);

$.getJSON('JSON/?php echo $file1; ?>.json', function(data1) {
    var chart1 = new Highcharts.StockChart({
        title: {
            text: '<?php echo $attribute1; ?> vs Time'
        },
        chart: {
            renderTo: '<?php echo $attribute1; ?>'
        },
        series: [{
            name: '<?php echo $attribute1; ?>',
            data: data1
        }]
    });
});
```
6. Conclusion

The project is available on the Internet at the URL satnet.fgcu.edu/~velosa. It is operational and is available for students, researchers and anyone who is interested in analyzing the data acquired and visualized from the 15-acre solar field on the Florida Gulf Coast University campus. The range of data are current, starting from April 1st 2010 at 10:45pm, and up-to-date within 15 minutes of the users access to the website.

The overall goal was accomplished, which was to develop a user interface that provided an easy-to-use format that a user could navigate easily and effectively. The efficacy of the server side software was also accomplished, by preventing excess use of storage required by the program. This was accomplished by a Java program that recursively deletes the contents of a directory in 12 hour intervals. The software is also real-time, updating the SQL database available locally on satnet.fgcu.edu in 15 minute intervals. There was also a lot of time spent trying to implement Amazon Web Services simpleDB, sqlite, web000 SQL server, and MongoDB before SQL server on satnet.fgcu.edu was used.

The project used a wide range of APIs and languages to accomplish its goals. The client side required HTML5/CSS, PHP, Javascript, Highcharts, jQuery, and JSON. The server Side required PHP, Java, and apache commons APIs.

This project has been a pleasure to work on. The developers hope that it will be used by Florida Gulf Coast University as either a research or information tool. It is also fully extensible and is recommended to future developers to investigate and integrate data mining technologies.
7. References


Appendix A

Code for SolarFlare class:

```java
package SolarFlare;

import java.io.File;
import java.util.Scanner;

/**
 * @authors Felipe Velosa, Vincent Giannone
 */
public class SolarFlare implements Runnable {

    public void run() {

        while (true) {

            try {
                // Navigates site
                HTMLNavigator website = new HTMLNavigator();
                // Connects and queries database
                DBManager database = new DBManager();

                // username and password to connect to solar ems
                String username = "ttalov@eagle.fgcu.edu";
                String password = "solarfgcu";
                String HTMLPage = null;

                // pages to navigate in sequence, last page is the target page
                String[] urlsToNavigate = {
                    "https://solarems.net/projects/36-fgcu-ab7/overview",
                    "https://solarems.net/projects/36-fgcu-ab7/data_sets/26"};

                // read attributes to retrieve values. This is a local text file
                Scanner AttributeNames = new Scanner(new File("AttributeNames.txt"));

                // login to website
            }
        }
    }
}
```
website.login("https://solarems.net/user_sessions",
"https://solarems.net/login",
    username, password);

// fetch last website in array "urlsToNavigate" website contains values
HTMLPage = website.fetch(urlsToNavigate);

String stringOfNames = "";
String stringOfValues = "";

// Parse through HTML page and the value for each attribute name.
while (AttributeNameNames.hasNext()) {
    HTMLParser parser = new HTMLParser(HTMLPage);
    String attr = AttributeNames.nextLine();
    double value = parser.getValue(attr);

    stringOfNames = stringOfNames + "" + attr + "" + ",";
    stringOfValues = stringOfValues + value + ",";
    // print attribute and value
    System.out.println(attr + ": " + value);
}

// remove the last comma (,
stringOfNames = stringOfNames.substring(0,
            stringOfNames.lastIndexOf(",")
        )
        stringOfValues = stringOfValues.substring(0,
            stringOfValues.lastIndexOf(",")
        );

// Prepare a SQL string to insert values into database
String SQL = "INSERT INTO "Values"(" + stringOfNames + ") VALUES " + "(" + stringOfValues + ")";

// Connect to the database
database.connect();
// Insert values into database
if (database.update(SQL)) {
    System.out.println("Database updated successfully");
}

// thread to sleep for 15 minutes
Thread.sleep(960000);

catch (Exception e) {
    System.out.println(e);
}

/**
 * @param args the command line arguments
 */
public static void main(String[] args) throws Exception {
    Thread t = new Thread(new SolarFlare());
    // this will call run() function
    t.start();
}

Code for HTMLParser class:
/**
 * @author Felipe Velosa
 */
public class HTMLParser {
    Scanner scan;

    public HTMLParser(String html) {

scan = new Scanner(html);

public double getValue(String attributeName) {

    Scanner byName = scan.useDelimiter("name">");

    while (byName.hasNext()) {

        String currentLine = byName.next();

        if (currentLine.contains(attributeName)) {

            Scanner byValues = new Scanner(currentLine);
            byValues = byValues.useDelimiter("value">");

            byValues.next();

            if (byValues.hasNext()) {

                String valueString = byValues.next();

                char[] c = valueString.toCharArray();

                String strVal = "";
                for (int i = 0; i < c.length; i++) {
                    if (c[i] == '0' || c[i] == '1' || c[i] == '2'
                        || c[i] == '3' || c[i] == '4' || c[i] == '5'
                        || c[i] == '6' || c[i] == '7' || c[i] == '8'
                        || c[i] == '9' || c[i] == '.' || c[i] == ',
                        || c[i] == '-') {

                        if (c[i] == ',') {
                            continue;
                        }
                    }
                    strVal = strVal + c[i];
                }
            }
        }
    }
}
} else {
    break;
}

} try {
    double value = Double.parseDouble(strVal);
    return value;
} catch (Exception e) {
    return -9999;

} }
}

System.out.println(attributeName + " NOT FOUND");

// no attribute name found, return dummy value
return -9999;