**Overview**

**Requirements**

**Development Methodology**

**Design**

**Implementation**

**Testing**

**Status**

**Critical Evaluation (future work/V2)**

**Bibliography**

**Tiers of use**

Programmer – designs and builds template system, no involvement once signed off

Client – Huw – sells to people

User – sets up and maintains their project: nodes, variables, etc

Guest – views data

**Client requirements**

This project was developed for a client, Acanthus Consulting. The client required a web-based system that could be sold as a service, allowing users to capture and store data from a wide range of remote monitoring systems in different locations. Potential applications for this technology were to range from logistics management to pollution monitoring to traffic control.

All potential applications required the inputted data to be displayed through a user-friendly interface requiring little technical skill to view. The client’s brief for the interface was to allow the user to select and view individual records and to be able to perform large-scale data exports.

Two test-cases were requested as proof of concept. The first was a remote environmental monitoring system, to be used in an academic project studying glacial microclimates. This project utilised a set of many separate monitoring systems to be spread across a wide geographic area. These capture a suite of environmental data, including temperature, pressure and humidity and light level, as well as a series of systems information such as battery and solar power voltages. All individual nodes are connected wirelessly to a central hub, which broadcasts the data to a server. The client required a central database for storing and retrieval of this information. The second test-case is a logistics management system for a milk delivery service.

A key early decision was to design a ‘relational’ rather than a ‘flat’ database. This was crucial to the expandability required by the client, who requested that the product should be ready to sell to clients ranging in scale: from those with very small requirements to clients with applications broad in scope and scale. In other words, it was requested that one service be able to provide for all client types without needing to be reprogrammed for each customer. Another reason for this decision was the database-design maxim, ‘columns are expensive; rows are cheap’. Using a flat database would have produced a very wide table with many empty columns in all but the largest projects, with no guarantee that the arbitrarily-chosen number of columns would be sufficient and not require reprogramming in future.

Data definitions – user needs to be able to set up project, identifying source of data and method in which it will be captured; set up how data is to be stored (i.e. what each variable in the incoming data stream corresponds to in real life); e.g. 23 is MPH or degrees Celsius; etc.

It was also a requirement that the user be able to register when the calibration of a long-duration instrument changes. This information had to be stored with a time-stamp indicating the relevant period for which that value was applicable as well as which data points are affected by each calibration period. In other words, the system must be capable of handling cases in which monitoring equipment is left on site for years at a time, with the raw data adjusted each time the equipment is recalibrated.

The client’s brief was for an interface that would allow users to see all data sources on a single overview screen, and then drill down into a specific record to see more detailed information about that data point. As this was to be a location-aware system, a specific requirement was to use a map to display the range of different monitoring locations used to capture data. These points were to be dynamically displayed using the inbuilt marker system integral to the Google Maps API. The client also requested that the webpage be able to display a detailed time-trace for a chosen variable. For example, a client might want to click on one particular location and be able to see temperature changes mapped over time. It was decided to display this data in a scatter-plot format using the Google Chart API. The Map and Chart APIs offered a simple, lightweight method of presenting data on the page, as these allowed for easy display of an array of GPS coordinates and environmental data.

A further requirement was for the system to be able to display data from static sources, where the GPS coordinates were provided as meta-data; as well as from mobile sources, where the GPS coordinates were provided as variables within the data stream.

**Program description**

**Development methodology**

This project was developed using a Feature Driven Development (FDD) approach. FDD combines iterative and incremental methodologies and is one of the components of Agile software development. This development method takes its impetus from deliverable project features. The main purpose of this approach is to deliver the software with full features required by the client on time. As John Hunt explains, ‘[a]daptive, iterative projects are more complex to control, and to plan, than more traditional, waterfall linear models’ (2006, p161). However, as this project was created as a solo enterprise, the management of time proved simpler to organise than other methods of Agile development including pair-programming. Working with this methodology meant that the design was led by asking questions such as how to best meet the client’s requirements and creating a logical order in which to do so.

The methodology began by analysing the project specification document provided by the client. A list of required features was drawn up and an initial project brief written prior to the initial client meeting. The aim of this was to allow a contrast to be drawn between the author’s initial view of the project and the client’s wishes, to minimise the possibility of misunderstandings and confusion. This proved to be very successful, as the project brief provided a blueprint for the initial meeting, giving the client a structured opportunity to correct inaccurate assumptions and identify differences in design.

The initial client meetings focussed on finalising the database structure. This was because the first priority was to get a working database around which the website could later be constructed. This corresponds to the tenets of FDD by identifying key features that should be completed as milestones in the project. The database was identified early on as the foundation for the project. In the event that small but fundamental changes are made to the database once more code has been written, the quantity of code that needs to be adjusted to accommodate this change is magnified. During several subsequent client meetings the database design was honed and a final specification including graphical representation agreed upon. The evolution of this design is further discussed in Section ??.

With an agreed database schema in place, the next step was to plan the project’s timescale. Due to the fixed nature of the final deadline and the requirement for a detailed write-up, an in-depth GANTT chart was constructed to plot interim deadlines from key feature deliverables. This can be viewed in Fig. ??? below.

**Design goals**

**Languages**

The primary decision that needed to be made was the language in which the program would be written. It was decided to use Java Enterprise Edition (EE). According to Kevin Mukhar *et al.*, '[e]nterprise applications solve business problems. This usually involves the safe storage, retrieval, and manipulation of business data [...]. Enterprise applications must deal with communication between remote systems, [and] coordinate data in multiple stores' (2006, p1). Alternative languages that were considered were Ruby on Rails and C#.

It was decided to use Java Server Faces (JSF) to build the webpages that would display the content dynamically loaded from the database.

JavaScript.

**Database design**

The decision to use Java as the backing programming language allowed JDBC to be utilized as the database manager. In initial discussions with the client, it was determined that the best way to approach the database design was to consider the user’s workflow through the program from first introduction. The user steps were decided upon as follows:

1. User registers their information;
2. User sets up a project;
3. User sets up the number of different node types;
4. User defines what instruments are on each node type and the order in which they are received from the system;
5. User defines the physical nodes based on Step 3;
6. Program stores the data for each node based on the information provided above.

This is the basic structure for a simple project, not allowing for input of any calibration information or management of duplicates and tags. These were added later to allow a more complex project to be managed over a longer period, as discussed below.

One decision that was made during early planning stages was how to implement the client’s requirement for mobile data sources. This required the system to return GPS data and process this separately from other variables. It was decided to add a Boolean flag to the Node table to indicate whether or not the device was static. This assumes that if the device is static it will not return GPS coordinates as part of the data stream. In addition, if the device is mobile and hence returning GPS information, a flag in the DataDefinitions table indicates which variables relate to GPS information. A possible alternative would be to store GPS data records as whole objects rather than separate variables; however, due to the client’s aforementioned request to streamline the database this method was deemed less optimal.

An early discussion with the client focused on this following question: What is the best way of taking a comma-separated variable and storing it, so that all data is stored in the most efficient way possible while still allowing that information to be reconstituted at a later date? In this respect, the first design was considered suboptimal:

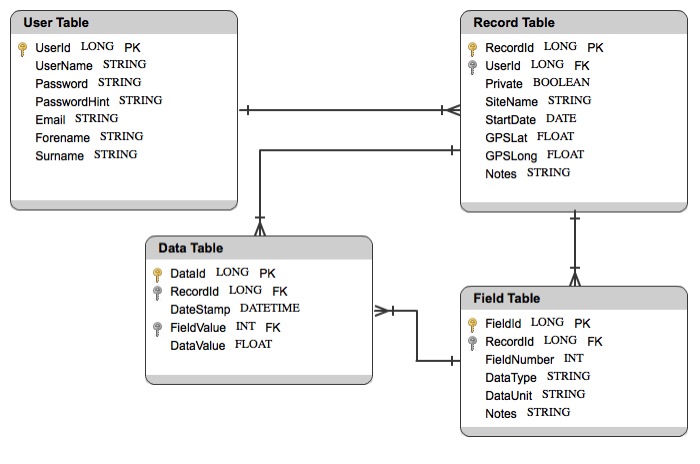
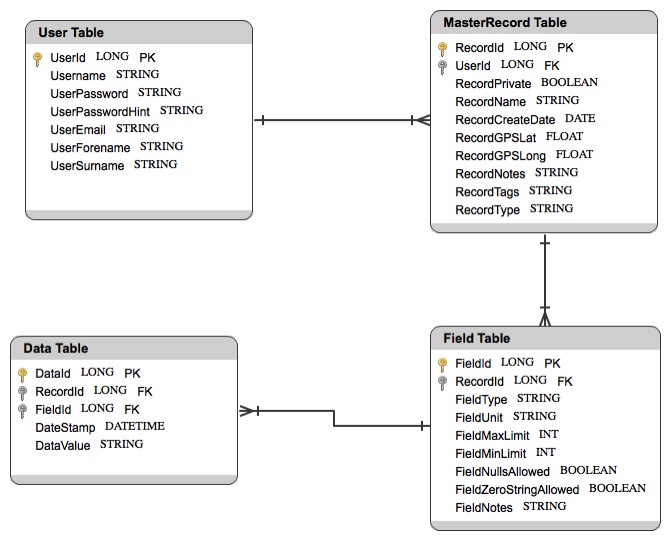


Fig. ??: Database design V1

Figure ?? above depicts an initial database design constructed prior to meeting with the client. The principle issue with this was the unnecessary relationship between the Data and Record tables. The design that was presented to the client at the first meeting was therefore a revised version of this, depicted in Figure ?? below:

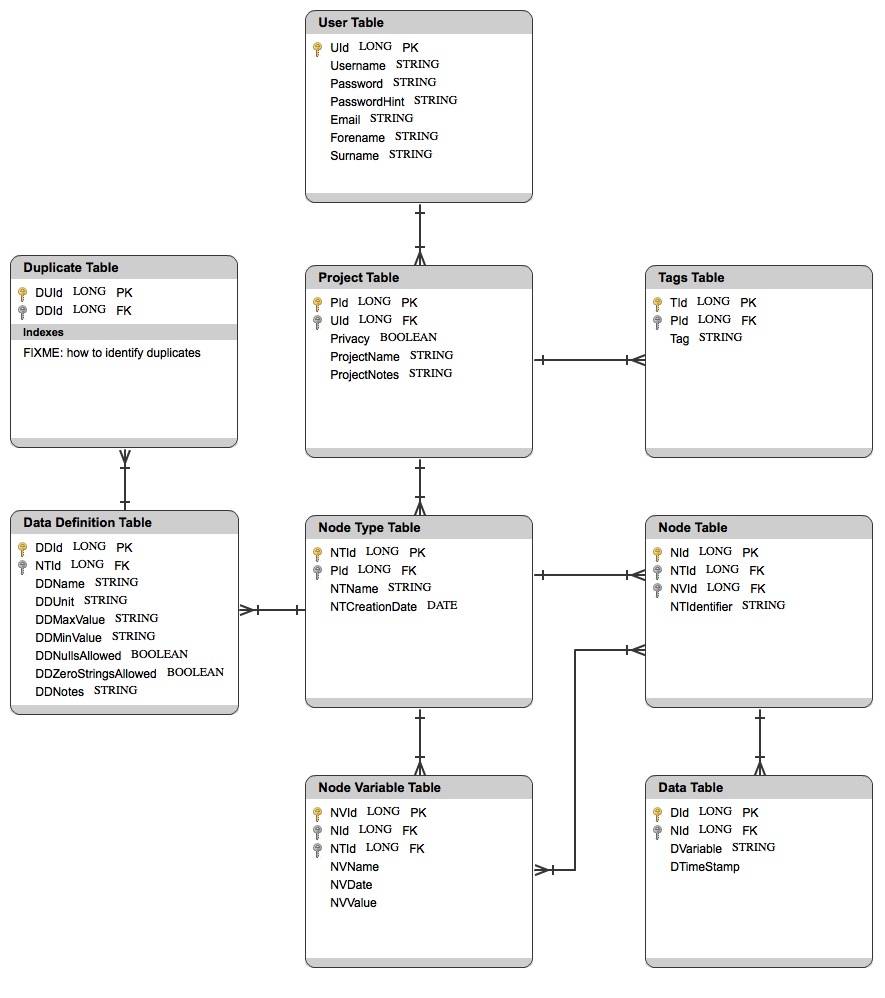


This design assumed that one user could have many records, and that each record could contain many variables. The variables would be stored in the Data table; the record information would be stored in the Field table; and the MasterRecord table would store information about the monitoring station itself.

Aside from the confusing nomenclature, the primary issue with this design was its inefficiency in terms of data storage. The client further identified that this would only allow each user to operate a single project.

This design was also considered to be unnecessarily wasteful of the user’s time. The first meeting with the client therefore asked the question of how to ease the user’s burden. For this, a hypothetical user was posed with 50 static stations, all with ten variables that required storing in the database.

One possibility that was discussed was to give each node many data definitions. However, this would not have provided a substantial time-saving, as each variable would still need to be entered 50 times, once for each station.



However, if you accept the memory penalty of having an additional table entitled ‘node type’, assuming that the 50 nodes are identical, the user only has to enter each variable each once; then each node can be set to have a specific node type, greatly reducing the user’s input burden. Small increase in size of database and incremenetal increase in processing time, but a substantial decrease in administrative burden placed on the user during set-up.

labour required by the user to set up their project

Image of pre-programming database design

??? Explain why this is better.

Flat database vs. really complex relational mapping – wanted to do something in between. Minimise the information in every table – minimize repetition = wasted columns and wasted rows.

It became immediately obvious during initial client discussions that a flat database would be completely inappropriate. Due to the integrated nature of Java EE for building web applications, it was decided to use a relational design as the entity framework. Object Relational Mapping (ORM) allows you to set out object classes in Java and then construct the database using that structure.

**Design evolution**

One of the aspects that evolved after the initial database design stage was the Node table. In the initial planning stage it was decided that there should be a ManyToMany relationship between this and the NodeVariable table. This was designed to allow many node variables (e.g. calibration records) to apply to a single instrument on one node, and also to allow the user to apply a correction factor (e.g. local timezone) to many nodes. However, it was determined during construction that the explicit generation of a ManyToMany relationship would be redundant, due to the object’s relationship to the NodeType table. As both Node and NodeVariable share a OneToMany relationship with the NodeType table, this acts effectively as a link allowing for navigation between the two objects.

A later meeting with the client identified a supplementary request: that the user should be able to manage duplicate data as it is transmitted from the monitoring system to the program. To this end, it was decided to include a Duplicates table. When two records are received during short order, certain variables will be identical (i.e. source package; destination package; etc), even if the record itself is not a duplicate. The Duplicates table needs to contain the information that can be used to identify when a record is a direct duplicate of another, rather than containing some similarities. For example, a hypothetical example that was discussed was a case where two records are identical in all fields with the exception of the time-stamp. The decision to include a Duplicates table was designed to allow the user to add conditions to the DataDefinition table, enabling them to set their preferred parameters by which duplicate data entries could be identified.

The original design envisaged that after the webpage had loaded, the user would identify themselves by selecting their User ID and then select their project. The webpage would then load up all the information relevant to that project and the user would be able to select a marker on the map and retrieve data about that location from the database directly to the graph. The initial design would have taken the user’s selection as input in a function call to a backing Bean, which would use that information to interrogate the database using a named query and then return all relevant data to the page. Unfortunately, this plan was based on a lack of knowledge of Javascript’s behaviour. It was initially assumed that data could be passed between Javascript and JSF in a manner equivalent to passing data between Java classes (i.e. synchronous data transfer). In actuality, because JSF builds the page before the Javascript runs, it was realized that this would not work as planned. In practice, were a user to request a specific variable to be output in graph form, this would require the page to be reloaded each time. In other words, the user would click on a marker, select the variable, and nothing would happen; or the entire page would need to be reloaded in order to display the result. One potential solution was to load all variables for all nodes simultaneously after selecting the project, and to store all data as a Javascript variable Array. This was originally chosen because of the relative simplicity of set-up. Storing the data locally would theoretically result in an increase in speed when selecting multiple variables, as well as offering the possibility of displaying multiple variables on the same graph simply. However, two drawbacks of this approach were identified. Firstly, for large projects, simultaneously loading a significant amount of data into the webpage was found to slow down the initial page load, resulting in a delay of ten minutes before the page could be viewed. Secondly, projects in receipt of real-time data would not be able to view any new data until the user had reloaded the page.

After consultation with the client it was decided to rework the project to use AJAX (Asynchronous Javascript and XML). The first requirement when changing the system to an asynchronous model was to rework the code in the backing Bean, so that rather than loading all data for all nodes it would only return data for the selected node and variable. This involved removing two Loops: one of which looped over every node in the database, and the other looped over every variable. These were replaced with input integers which indicated which node and which variable should be returned. One identified issue was that testing this required hard-coding temporary variables in the Bean call whilst the rest of the AJAX was being developed. This was necessary during the development process to ensure that the chart object continued to display data correctly. A regularly-encountered problem while developing the AJAX component was that a change would be made in Javascript or JSF and would result in the chart object no longer being rendered. Therefore, stock data was required to ensure that there was always data available to be displayed for testing purposes.

The principle development of the AJAX component involved three stages:

**Stage One:** Firstly it was important to build an HTML AJAX call using the f:ajax component of JSF as a proof of concept. For this the input was performed manually and the output passed to a text field. This was done to ensure that the AJAX call was working correctly before attempts were made to automate and integrate it with the Map and Chart components. The first issue encountered in this process was that despite the reduction in run-time of the backing Bean method, returning all the variables simultaneously was still prohibitively time-intensive. Therefore the return was limited to ten variables. A longer-term solution… ???. Secondly, it was realized that the JSF AJAX components need to be a part of the active website in order to be rendered. However, the user does not need to interact with them, as this would detract from the intended simplicity of the design. The solution was to use CSS to collapse the containing component so that they become invisible to the user. The second major issue in this stage surrounded the research undertaken before building this component. Reference to forums such as StackOverflow found that the majority of examples provided did not use JSF, and were therefore set up to return webpages rather than returning raw data. The solution suggested by early research was to use Javascript and JQuery to perform the AJAX call. This was unsuitable, as in every example component webpages or lists of static data were being returned; whereas this project needed to return dynamic data from a backing Bean. Eventually it was realized that the f:ajax component of JSF could be used to trigger an AJAX call based upon input data and to render the output asynchronously on the webpage. Initial efforts to implement this using two h:inputText fields, one for the node and one for the variable, only returned the change in variable. Efforts to resolve this included sending the data as a comma-separated string in one inputText, and setting the f:ajax component’s ‘execute’ field to accept both inputText fields comma-separated. Neither of these solutions was deemed acceptable: the first resulted in the need for additional coding to concatenate both variables; while the second produced formatting error messages. The ultimate solution chosen was to use a space-delimiter rather than a comma.

**Stage Two:** The next challenge was to pass the data from the output field or the backing Bean itself into the Chart object. The first step in this process was to split the Draw Chart call out from the action listener in the info window. Because AJAX is asynchronous, the call happens and is followed by a delay before the data is returned. This required the implementation of two separate functions: one which calls the AJAX and another which listens for the AJAX response. Therefore, the Draw Graph method call needed to be separated from the action listener in order to later implement the AJAX listener.

**Stage Three:** Once the Draw Graph function had been separated, a function could be written to listen out for the return of the AJAX data. This was undertaken by having a method call which would listen out for changes in the AJAX data.status field. This field has three possible options: ??, ‘completed’ and ‘success’. As the backing Bean data is persisted across the session, receipt of a ‘status===”success”’ message indicated that the data could be retrieved from the backing Bean, stored to a global variable in Javascript, and the Draw Graph method could be called. Initial testing indicated that the graph was being drawn successfully, but only on the first instance. In other words, the user was able to select one node and variable and to view the data in the chart, but would not then be able to change the chart to reflect a new variable.

**Stage Four:** By this stage, the project included a separated call-and-response listener; a successful AJAX call; and a graph which automatically updated its data based on user input (albeit once). The challenge now was to automate the AJAX inputs based upon the user’s selection of data in the Map. Initial efforts to set the text field values from Javascript were unsuccessful. The command ‘document.selectElementbyId’ can normally be used to identify a JSF component so that changes can be made to it. Unfortunately, in this case the AJAX component needed to be included within an h:form component and so a different solution had to be identified. Fortunately, ‘selectElementsbyClass’ was discovered to successfully return an array of components sharing a specific class. Therefore the node and variable fields and the select button were adjusted to be of the class “ajax”. Then the individual components could be identified through their ID and the data could be set correctly for the first two and the button clicked when the user selects from the drop-down menu.

**Stage Five:** The next stage was to address the issue of the data only updating once. It was discovered that the problem persisted after all the above changes had been made. Research indicated that nesting one h:form component inside another h:form component will lead to errors. The solution was to separate the two forms, resulting in correct repeated operation of the asynchronous graph updates.

owvH

A connected issue was encountered due to the need to store session information in JSF. It was important to the client that different users have the ability to operate the program, and so information on User and Project needed to be temporarily stored using a login system. Two storage options were discussed with the client. These were the use of cookies versus the use of server-side data storage as scoped variables. It was decided to use the latter, as this reduced the amount of data that needed to be transmitted between server and client side and therefore reduced the burden on the user’s internet connection during the initial page load, as described above.

During development of the map display, two issues occurred. Firstly, the design called for the full array of project nodes to be displayed on the map at once. This was designed to allow the user to select data from many different nodes with ease. However, it was noticed that only one node was being displayed. Secondly, in addition the displayed location was significantly different to the GPS coordinates that had previously been entered. Upon investigation it was discovered that all nodes were actually being displayed but that each had identical coordinates, rendering all but the top node visible. Interrogating the database indicated that the GPS coordinates of all nodes had been rounded off to 0 decimal places, resulting in a location somewhere around Merthyr Tydfill. Upon re-entering the correct data, the same problem occurred. Further investigation revealed that the variable had a Scope set to 0. The original entity design had used a Float variable for the GPS coordinates. However, a lack of specification of the precision of this Float resulted in the database using a default value. The solution was to change the type from a Float to a BigDecimal, and use the ‘@Column’ annotation ‘(precision=12, scale=8)’ (Entities.Nodes, Line 45). This supports all forms of decimal GPS coordinate and thereby solved the issue.

This change, undertaken after significant development work had already been performed, required the database to be rebuilt from the entity classes. This was because the very structure of the database had been changed. It was initially attempted to force an update of the existing database, but this solution was ineffective. A fresh persistence unit was therefore built with a new database, which solved the problem but with the downside of requiring several hours to be spent reentering data. A positive outcome, however, was the identification of a previously unknown bug that caused the manual input of test data to fail. It was discovered that early attempts to get some data stored in the database for development purposes had resulted in a file upload object that was highly unstable, hard-coded to a specific file, and would only upload a single line of data. In addition, it was set up to receive data from a .RTF file which had eight lines of header information inherent to the design of the .RTF file specification. The chosen solution at the time was to pick a line close to the end of the file and only enter that line of data. Rebuilding the database provided the opportunity to replace this code with a temporary solution for .RTF files, which ignored the first eight lines of header and the last line of footer, allowing the entire file to be uploaded into the database in a single action. The learning outcome from this was that whilst quick fixes might be useful during the initial tentative development stages, the work required to improve upon early shortcuts is more extensive than methodically implementing a correct solution the first time around.

**Interface**

Using the Google Maps API permitted the use of the marker and info window system for displaying nodes and variables. The requirement to be able to display different users’ projects required that markers, and their associated info windows, be generated dynamically. This was because it could not be known what the location and quantity of nodes would be for each new project. The data display was originally envisaged as a drop-down menu. Clicking on the menu would show the range of available variables for that node. For this, an action listener would have been used to identify when a new variable was selected by the user and to trigger the redrawing of the graph based on that variable. However, during the build process a bug was discovered whereby the action listener was not sensitive to any drop-down menu change, regardless of marker. The issue was discovered to be a lack of allocated component in the HTML. In other words, there needed to be one select menu of the required size for each marker in the HTML, completely separate from the map, and the output then needed to be cloned into the infowindow. This then had to be hidden from the user. Before this solution was discovered, an attempted workaround to the insensitivity of the action listener was implemented. This initial workaround was to dynamically generate buttons representing each variable in the info window, each of which used the onclick function to trigger the redrawing of the graph. The fact that the onclick did not work when generated inside the infowindow suggested that there was a fundamental flaw in the core design of having an action listener linked to a unique component in the infowindow. This was reinforced by spike work, suggesting that an action listener could easily be hard-coded to a fixed component in the HTML. This led to the realization that the initial design of a drop-down menu could be possible as long as it existed on the page as a separate entity to the map interface. The original design of a drop-down menu was therefore retained and the alternative list of buttons dropped.

An envisaged future implementation would allow for the drawing of multiple variables on the same graph for comparison purposes. In this system the user would select the variable in which they were interested from the drop-down menu.

The original intention was to craft a chart object using HTML Canvas, allowing for dynamic adding of data as it was received. However, the processing requirements for a large graph precluded this method, and the Google Chart API was selected as a superior alternative after discussion with the client. This was because of the flexibility offered by the Google Chart API when inputting data and the ability to dynamically change the data types based upon the user’s stored parameters.

Discussion of what the chart offers, how it was implemented, reset, bugfixes including javascript break for axis === 2, use of empty return statement, planned improvements(dashboard) that can be rewritten if I get time to inlcude.

**Data input**

The current method of uploading data requires the manual addition of a text file. Due to the format of the test data, currently only CSV files are supported. This design restriction was undertaken to simplify the data input programming.