mpdb-client

Anthony Waters

February 25, 2008

Contents

Part I

Introduction

The part of the project mpdb-client is responsible for creating, display, and updating the user interface of the application. All of the source code within this project is converted to JavaScript through the use of Google Web Toolkit (GWT).

Chapter 1

Bulk Upload

Bulk upload is the process in which a document in a preset format is parsed for data, basically a scientists adds a lot of samples to a spreadsheet and then the application parses the spreadsheet to add samples. This is beneficial because it allows the scientists to circumvent the process of adding the samples individually by hand. The process of performing a bulk upload can be summarized into four steps3

1. Uploading the document
2. Parsing the document
3. Saving the objects in the database
4. Reporting the status of the operation (success/failure)

1.1 Uploading the Document

In order to transfer documents from the client to the server a file upload needs to occur. This is accomplished with a GWT widget named a `FormPanel`, this allows for sending POST request to the server. Another situation that this is used in is in upload images to the server. A very simple interface for testing bulk upload is located in the package `edu .rpi .metpetdb .client .ui .bulk .upload` the name of the class is `BulkUploadPanel`. If you look at it you will notice how simple it is, just a file upload control. The interesting thing to note here though is where the `FormPanel` posts to, which is `fp.setAction(GWT.getModuleBaseURL() + "/bulkUpload");`. What this means is it post to the bulkUpload servlet on the server, which brings us to the next section.

1.2 Parsing the Document

The post request is sent to the bulkUpload servlet which is located in the package `edu .rpi .metpetdb .server` within the class `BulkUploadServlet`. This is a simple servlet whos only function is to save the file on the server, and then based on the type of bulk upload it passes the file onto the parsers located in the package `edu .rpi .metpetdb .server .bulk .upload`. Parsing the files is a fairly simple task because there is a library already available from Apache called POI, which stands for "Poor Obfuscation Implementation". The part of the libary that is used for handling the spreadsheets is called HSSF, which stands for "Horrible SpreadSheet Format" (it is refering to Microsoft Excel in this case). A simple tutorial on how to use POI and in particular HSSF is located at http://poi.apache.org/hssf/how-to.html. An example is also located in the source file `SampleParser.java` in the package `edu.rpi.metpetdb.server.bulk.upload.sample`. It works exactly like going through a table, you go row by row and column by column. So in order to parse the spreadsheet one would first read the first row, which contains the column headers, then subsequently read every other row parsing the data and determining what it is based on it's column header. How to actually save that data is discussed in the next section.

1.3 Saving the objects in the database

In order to explain this section I will use a Sample object as an example, therefore the user uploaded a spreadsheet that contains just samples. The java bean for the sample is located in the class `Sample` in the package `edu .rpi .metpetdb .client .model` this contains all of the properties for a sample that are read and written by the database. With that in mind in order to save the data to the database the following has to happen

1. Create a new sample object
    - `final Sample s = new Sample()`
2. Set the properties of the sample based on the data
    - `s.setAlias(‘‘My Lovely Sample’’);`
3. Save the sample to the database
    - `saveSample(s)`, located in `SampleServiceImpl`

The next step is to notify the user of the result of the operation.

1.4 Reporting the status of the operation (success/failure)

To send back a message to the client about the status of the operation you could do the following

1. If there was an error
    - throw an exception that will get passed to the client
    - `throw new InvalidFormatException`
    - the exception will contain the necessary details to will be displayed to the client about what caused the error
2. If it was successful
    - Return a string telling the user that is was successful
    - Add in things like what was added and links to them

Chapter 2

Adding/Editing Objects

Adding and editing objects forms the basis of the database therefore their functionality is vert important. The type of objects that can be edited include

- Grids (Subsample Maps)
- Images
- Mineral Analyses
- Projects
- Sample
- Subsample

As a basis for describing the procedure of how adding and editing objects occurs I will use a Sample because it represents the most complex object to date.

2.1 Adding Objects

The start of the interface for adding Samples is located in the class `SampleDetails` that is within the package `edu .rpi .metpetdb .client .objects .details`. Within this class there are a couple things that are unique to this class

1. `private static GenericAttribute[] sampleAtts`
2. `private final ObjectEditorPanel p_sample`

Basically the way it works is all of the `sampleAtts` are layed out on the `p_sample` widget.

2.1.1 What are these GenericAttributes?

The `GenericAttribute`s are composed of two things

1. the attribute itself, i.e. `TextAttribute`
2. the constraint, i.e. `MpDb.doc.Sample_alias`

2.1.2 The Attributes

The attributes is the actually GWT Widget that is placed in the DOM and rendered to the browser, therefore, if you need to change something UI related it is in the attributes. If you look at the `edu .rpi .metpetdb .client .ui .input .attributes` package you will notice that there are regular attributes in the parent package and attributes that are in the `specific` package. Attributes that are in the regular package are very generic, meaning that they can be used in any object along as the constraint is correct. A good example of a regular attribute is the `TextAttribute`. The primary purpose of the text attribute is to allow for single line free text to be entered, as you have guessed, this functionality is required in many of the objects. As opposed to regular attributes, the specific attributes are very specific in functionality and are very hard to abstract. A good example of a specific attribute would be the `MineralAttribute`, whose prime purpose is to allow the user to select various amounts of minerals from a tree. The beauty of this specific attribute is the fact that it makes use of a regular attribute, the `TreeAttribute`. In describing the attribute I will refer to the `TextAttribute` as the example.

Creating an Attribute

First off attributes are **required** to extend a `GenericAttribute` , along with extending that class a few methods need to be implemented

- constructor that calls `protected GenericAttribute(final PropertyConstraint pc`
  - This is necessary because it initializes the `GenericAttribute` with things like the label (which is displayed to the left of the attribute). An example of a label would be "Sesar Number"
- `public abstract Widget[] createDisplayWidget(MObjectDTO obj)`
  - Whatever that is returned here will be added to the DOM when an object is being viewed. For example with the `TextAttribute` this method returns just a `Label` because it is used to show data. The reason that this methods returns an array of `Widget`s is because for each widget in the array a new row in the display table will be added.
- `public abtract void set(final MObjectDTO obj, final Object value)`

  - When the user clicks Save the new object is passed in as the first parameter, and then the value for that object is passed in as the second parameter. Usually this method just calls `mSet` which sets the value in the object itself. For more understanding look at the `mSetGet` method of any of the Java Beans, i.e. `Sample`.

while the following methods are generally implemented as well

- `public Widget[] createEditWidget(final MObjectDTO obj, final String id)`
  - This method is very similar to `createDisplayWidget` however instead it is used when adding/editing objects as opposed to viewing them. In the case of the `TextAttribute` this method would return a GWT `TextBox`.
- `protected Object get(final Widget editWidget)`
  - As mentioned before when the `set` method is called the second parameter is the value to set, well the `get` method is responsible for getting this value from the `Widget` that is being edited. For example with the `TextAttribute` the `get` method would be called with the `TextBox` passed in as the editWidget. Then the method proceeds to extract the text from the `TextBox` and returns it. In more advanced attributes usually editWidget is never used and the attribute itself keeps track of its value through alternate means.

More advanced attributes have more methods take a look at the `TreeAttribute` if you want to see them.

2.1.3 The Constraints

The constraints are for validating the data that is contained within the attribute. For example the constraint would be responsible for making sure that the user entered an integer in a whole number only field. There are more advanced instances than that, like verifying that the user chooses an element is in a predefined list. In order to explain constraints better I will use the `StringConstraint` in the package `edu .rpi .metpetdb .client .model .validation` [1]

Creating a Constraint

Constraints are easier to create than attributes in the sense that they are not required to extend any methods. However to make the constraint actually do something useful some methods are helpful to extend. Before the methods are explained it is important to note that the each constraint needs to extend a `PropertyConstraint`, which by default verifies only that required attributes have non null values.

[1] It is located in the mpdb-common project

- `public void validateValue(final Object value) throws ValidationException`
  - This method is responsible for validating the passed in value. If the value fails validation this method should throw an instance of a `ValidationException`. In the `StringConstraint` the purpose of `validateValue` is to verify that the `String` meets the requirements of minimum length and maximum length [2] [3] .

2.1.4 ObjectEditorPanel Who?

TODO

[2] It also verifies that the `String` is not null by using the superclass method `validateValue`

[3] You may be wondering where these minimum and maximum length values come from, well so do I. Actually these values are obtained from the server and is out of scope of this document.