Working in the

MemexPlex Code Base

**Introduction**

The purpose of this document is to familiarize a developer with the MemexPlex Application Code Base by walking them step by step through a fully functional screen for a web application accessed through a web browser that performs transactions with a database.

**Prerequisites**

Before working through this document, the software developer should have accomplished the following:

1. A basic understanding of the PHP programming language.
2. A basic understanding of object-oriented programming concepts and principles.
3. A beginner-level familiarization with Extensible Markup Language (XML), Hypertext Markup Language (HTML) and how it relates to the Document Object Model (DOM), and basic understanding of web development.
4. A basic understanding of JavaScript with applications to DOM manipulation and AJAX requests.

**Setting up the MemexPlex Development Environment**

1. **Server2Go Setup**
2. **Manual Setup**

**Application Setup**

**Database Setup**

20110101memexplex\_db.sql

**Create a New Project in Your IDE (Netbeans)**

Set up a new project in Netbeans by clicking **File>New Project**. In the **Choose Project** window, select “**PHP Application with Existing Sources**” and click “Next”. For the **Name and Location**, browse to the “**memexplex/**” folder, name the project “**memexplex**” and click “Next”. On the Run Configuration screen, make sure the project URL reflects your PHP server’s port and click “Finish.” Right-click on the “**memexplex**” project and select “**Set As Main Project.**”

**Create a New Project in Your IDE (Eclipse)**

**The Development Environment**

**MySQL Database**

The MemexPlex database strives to maintain the [Rules of Data Normalization](http://en.wikipedia.org/wiki/Database_normalization) in order to eliminate redundancy and improve data integrity. Currently the degree of normalization stands at [Fifth Normal Form (5NF)](http://en.wikipedia.org/wiki/Fifth_normal_form); however, through the proper application of key dependencies and table rules, it should not be difficult to attain [Domain Key Normal Form (DKNF)](http://en.wikipedia.org/wiki/Domain/key_normal_form) in the near future.

We recommend [PHPMyAdmin](http://www.phpmyadmin.net/) as a free, user-friendly, open-source PHP application for managing your MySQL database. If you are

**MemexPlex Database Schema**

****

**URL Masking (Mod\_Rewrite)**

Open the following file in a text editor:

.htaccess

Add the following entry to it:

**<Location /practice>**

 **Allow from all**

**</Location>**

**<Directory "C:/Program Files/Apache Software Foundation/Apache2.2/htdocs/practice/">**

 **Options -Indexes FollowSymLinks**

 **AllowOverride All**

 **Order allow,deny**

 **Allow from all**

 **RewriteEngine on**

 **RewriteRule ^(application|framework|log) - [L]**

 **RewriteRule ^(.\*)/(.\*)/(.\*)$ index.php?application=$1&pageCode=$2&$3**

**</Directory>**

This modification to the Apache configuration file allows for [URL Rewriting](http://httpd.apache.org/docs/1.3/mod/mod_rewrite.html), where a URL entered into the web browser several directories deep is read by a file in the root directory, which reads get variables from it. So the URL:

http://localhost:4001/memexplex/MemeList/

is interpreted by the index.php file as:

http:// localhost:4001/index.php?application=memexplex&pageCode=MemeList

Test the installation by opening a web browser and navigating to the directory on your local server:

**http://localhost:4001/memexplex/MemeList/**

The “MemeList” page should be displayed. This is because the framework reads the pageCode “MemeList” and the application “memexplex.” If either of these variables are missing, the framework will default to the memexplex application and the MemeList page.

**Overview of the Framework and Application-Specific Directories**

The MemexPlex Integrated Application Development Environment consists of two main directories “**Framework**” and “**Application**”. The Framework directory consists of the core functionality universal to all applications in MemexPlex, and, as such, contains files that should only be modified by software developers directly assigned to managing Framework files.

The Application directory contains subdirectories specific to applications. If you wanted …

**Framework Directories**

Although only a few developers will be assigned to managing the Framework files, it is important to understand the functionality these files provide and how they manage the application flow.

**index.php –** located in the root directory, this file is the core of the “[Front Controller Pattern](http://en.wikipedia.org/wiki/Front_Controller_pattern)”, which manages the flow of the application at a single point, ensuring those functions and classes necessary for all MemexPlex applications are loaded, such as initializing error handling, the user session, and determining along which path to direct the user based on the page code they are attempting to view.

**api –** this folder contains files the framework accesses through AJAX calls, such as the **communityMessage.php** and **processForm.php**, and functionality not contained within classes, such as the **autoloadClass.php**.

* **autoloadClass.php –** this function creates an array of all directories and files under the classes directory, which automatically loads the appropriate file when a new class is instantiated.
* **report.php, processForm.php, setSessionVariables.php –** these files respond to AJAX calls from the client with services such as returning the appropriate HTML tables of reporting data, accepting inputs submitted from an online form and sending them to the business layer for transactions with the database, or setting user session variables when changes are made in the interface.

**classes –** this directory contains all of the files for all of the classes loaded by the **autoloadClass.php** file. The classes are grouped by what application layer they fall into:

* **business –** business entities (ie. personnel, aircraft, other assets), business rules (data validation), groups of entities necessary to build a screen, and groups of business rules and entities into components.
* **control –** classes to manage functionality within the framework and provide utilities, such as user sessions, environments, error handling, exceptions, time, xml, constants, and xml utilities
* **persistence –** classes for managing data that must be maintained between user sessions, such as data stored in a database, cache, or XML file.
* **presentation –** classes that manage presenting data to the user, currently HTML for web presentations. Standardizes presentation with components to build application headers, tables, form elements, menus, and javascript programming exhibited on the front-end.
* **system –** server-specific utilities, database event handlers and listeners.

**config –** ini and xml application configuration files for defining server environments, database connection details, exceptions, and screen descriptions.

**css –** cascading style sheet files. To maintain a consistent look and feel for Almis, all pages use **almis\_style.css**; however, some javascript functions require additional css files, so those are stored here as well.

**images –** image files for the application, such as buttons, USCG insignia, and browser icon.

**javascript –** generic javascript files that any screen might make use of, such as date functions, adding rows, dynamic selectors, turbo search, and AJAX calls.

**tools –** application tools, such as APC cache and tools to read log files generated by the framework and application.

**Application Directories**

Beneath the Application directory, there will be additional directories assigned to each application using the framework (currently only MemexPlex).

**classes –** these are granular to the component level within a specific application, grouping the classes by their business function within the application, and are loaded automatically by the autoloadClass.php file. For instance “Training” and “Operations” might be subdirectories under the application directory, with the following directories listed beneath them:

* **business –** business entities (ie. memes, references, schemas), business rules (data validation), and groups of business rules and entities into components.
* **persistence –** classes for managing data that must be maintained between user sessions, such as data stored in a database, cache, or XML file.
* **presentation –** classes that manage presenting data to the user, currently HTML for web presentations. Standardizes presentation with components to build application headers, tables, form elements, menus, and javascript programming exhibited on the front-end.

**config –** configuration files for describing specific screens in the application, the datasource specific to the application, and the menu for the application

**javascript –** granular to the component level, these are the javascript files specific to a screen or component in the application

**Tiered Application Development**

The MemexPlex Applications Framework is designed based on the [Model-View-Controller (MVC) Architectural Pattern](http://en.wikipedia.org/wiki/Model-view-controller), which separates business logic from presentation and data persistence, allowing developers to work independently on different tiers with minimal impact on other layers. This way, database developers can write classes to interact with specific business domains in the database, while, simultaneously, web developers can lay out screens based on user requirements.

This architecture also introduces a new, third-tier to MemexPlex Applications development, the Business Layer, where all logical business entities and rules are maintained, providing a single location for gathering all business requirements. The business layer also serves as the intermediary between the persistence and presentation layers, allowing changes to one without impacting the other.

Developers will notice that in each business domain-specific directory in his or her application under the classes folder, they will find a persistence, presentation, and business directory, which aligns with the model, view, and controller paradigm.

**Business Layer Development**

For our screens, we will be dealing with the business of managing Memes. So begin by opening the “**Meme.php**” file in the **“/application/memexplex/classes/practice/business/entity/**” folder in your preferred IDE:

**/application/memexplex/classes/business/entity/Meme.php**

Take some time to read its contents. The comments provided between the “/\*\*” and “\*/” are [PHPDoc Blocks](http://phpdocu.sourceforge.net/howto.php), formatted in such a way to make the application self-documenting. It is a standard practice to include these comments in order to describe the purpose and functionality of your code to other developers as well as enhancing the functionality of the IDE’s [IntelliSense](http://en.wikipedia.org/wiki/IntelliSense).

Note that the name, value, and date properties for the class are protected. Further down in the code, you will find public methods **getName()**, **getValue()**, and **getDate()**. This standard is in order to hide the implementation details of the class from the outside. Encapsulating these details in this manner allows for easily modifying them with minimal impact to the outside objects that rely on them.

Often a class will also have a series of public methods like **setName($name)**, **setValue($value)**, etc, which allow for setting the value of these properties as well. In our example, we have placed the setting of the object properties into the **\_\_construct($name,$value,$date)** method to have them set at the time the object is instantiated like so:

*$meme = new Meme(‘foo’,’bar’,’01/01/2010’)*

Should it be necessary to modify these values later in the programming logic, set methods may be added to the class in order to accomplish this.

Finally, note the two methods **compare()** and **compareDates()**. These methods compare the properties of two **FooBar** objects and return 1 or 0 depending on the result. The purpose of these methods will come into play in the next, the “**MemeList.php**”:

**/application/memexplex/classes/business/entity/MemeList.php**

Open the **MemeList.php** file. Note that **MemeList** extends the **ObjectList** abstract class found under the Framework directory structure, which, in turn, extends the [ArrayObject](http://php.net/manual/en/class.arrayobject.php) class intrinsic to PHP. This allows **MemeList** objects to function just like arrays, an iterable collection, but extended to include validation and sorting methods.

The protected function **validateItem($item)** is triggered when an item is added to the list. If this item is not an instance of a **FooBar** object, then the **FooBarList** will throw an exception and return false.

The sort() and sortByDate() functions sort the list by accessing the compare() and compareDates() methods provided by the FooBar objects to compare the properties of the objects in the list and sort them appropriately. Note that sorting occurs in the list object *and will not be present in the database SQL query.*

The **FooBarProcessForm** class packages all changes into **FooBarDelta** objects:

**/application/memexplex/classes/practice/business/entity/FooBarDelta.php**

This class extends the **FooBar** class and refers to this class’ **\_\_construct()** method when setting the properties contained within that class. This way, any validation on those properties is maintained. The **FooBarDelta** class adds an addition **$flag** property, which indicates whether the **FooBar** is meant of insert, update, or delete when it goes to the **DataAccessComponent**.

**/classes/practice/business/entity/FooBarDeltaList.php**

Just like the **FooBarList**, this file encapsulates a list of **FooBarDelta** objects.

**/classes/practice/business/component/FooBarBusinessComponent.php**

This class encapsulates the functionality of obtaining all objects involved in executing the business of FooBars from the persistence layer where they are stored and for submitting object data lists modified by the client to the persistence layer for insert, update, and delete transactions.

Our example only involves a single list of objects; however, other objects could be assembled here if part of a logical grouping that would not be split apart for use elsewhere. A business component class may also access more granular business component classes if appropriate.

If these objects were stored in cache, then the logic to access them from the cache would be encapsulated here (see **QualificationBusinessComponent.php** for an example).

The **getFooBars()** method access the persistence-layer **DataAccessComponent** to retrieve the **FooBarList** object and return it to the requesting object. The **saveFooBars()** method accepts a **FooBarDeltaList** object, begins a database transaction, submits the delta list to the persistence layer’s **DataAccessComponent**, rolls back the transaction on fail, and commits the transaction on success.

**/classes/practice/business/businessobjects/FooBarBusinessObject.php**

This class encapsulates the functionality of obtaining all of the business component objects for a business domain, transforming them into XML for the presentation layer or any other client.

**/classes/practice/business/pageobjects/FooBarPageObjects.php**

Note that this class implements the **PageObjectsInterface**, meaning it must provide a public function **getData()**. This method accesses all of the **BusinessObject** classes necessary for building the user interface. In our example, this is only the **FooBarList**, but **getData()** could also load user data, data for select fields, or other elements needed for the user interface, but not confined to a single business object or component.

If variables from the client are needed to customize the data returned from this class, then these would be defined as protected properties of the class, settable through the constructor or setter methods.

**Persistence Layer Development**

**/classes/practice/persistence/component/FooBarDataAccessComponent.php**

The **FooBarDataAccessComponent** encapsulates all of the SQL interactions with the database to get the data to populate **FooBar** objects through the **getFooBars()** method and save data from the **FooBarDataList** object.

Note that the save and get methods of this class both invoke the static function **IngresDatabase::connectToIngresDatabase()**. If there is already a connection to the **IngresDatabase**, the application will not open a new one.

**Presentation Layer Development**

**The Page Code**

Everything in the presentation layer is keyed off of the **PageCode**. The **index.php** front-controller in the **practice/almis** directory gathers the **PageCode** from the URL in the web browser window and sends it to the presentation layer, where it is used to access the appropriate third-level menu, main body content, and javascript. The **index.php** also uses the browser URL to determine which application the framework is working with, like so:

*http://server-name/almis/application-name/page-code/*

An example of what the URL for our FooBar page example would look something like:

 *http://localhost/almis/eal/FooBar/*

The “eal” portion of the URL tells the framework to access EAL-specific directories and configuration files, and the “FooBar” portion of the URL tells the framework to access presentation-layer classes specific to the “FooBar” web page.

The presentation layer was architected with the requirement that web pages be standardized in their appearance, XHTML compliance enforced, and modifiable through data rather than editing HTML directly to streamline maintenance development. To achieve this, the presentation layer relies on XML configuration files to describe the contents of web pages:

**/config/pages/FooBar.xml**

At the top of this file, you will find the following XML nodes:

|  |
| --- |
| <title>FooBar Records</title><security>aircrew\_training</security><javascript> <script>switchContent</script> <script>FooBar</script></javascript> |

The **title** value is the title that will appear in the header for the page. The **security** node in the above example references the *page\_code* value in the database’s *page\_* table for the page in the security model. The **javascript** node lists all .js files to be included in the page. While the goal for web development will be to have the appropriate javascript included automatically when certain functionality is called in the framework, there will often be page-specific javascript tailored to the unique circumstances of the page.

Note the **forms** nodes in this configuration file, the children of this node both define how the form will be written in the web browser via the presentation layer and how it will be processed by the control layer. The following is an example of the properties of a Form Configuration file:

|  |
| --- |
| <*FormName*> <recordDataXpath> *xpath* </recordDataXpath> <formfield> <label>*label*</label> <type>*type*</type> <id>*name/id*</id> <valueXpath> *xpath* </valueXpath> </formfield></*FormName*> |

**FormName:** this is what the FormTable object will reference to build its source.

**recordDataXpath:** the xpath for the record set that the form will be populated with data, this can be used in a loop to iterate through rows of data.

**formfield:** a container tag for the set of attributes necessary to describe each form field.

* **label:** the label for the form field that will be displayed to the user in the table header or to the left of the field.
* **type:** the type of form field (ie. Select, Checkbox, Hidden, etc.)
* **id:** the form field’s name and id values, used by page JavaScript and form processing.
* **valueXpath:** the xpath where the current persistence-layer value of the FormField is located. This xpath should be relative to the recordDataXpath location.

For a more in-depth explanation of how the page configuration files describe HTML objects and are interpreted by the framework, please see the document **PresentationLayerDevelopment.doc**.

**/classes/practice/presentation/htmlcontent/HtmlContentFooBar.php**

This file represents the main content for the screen. Tables of data to be viewed by the user and forms to be edited. Using the [PHP SimpleXML](http://php.net/manual/en/book.simplexml.php) object provided by the page objects class, the **HtmlContent** object will define the form, block, and table components that make up the page. The strategy here is to create reusable blocks of html code, which will standardize presentation and interface functionality within the application.

The **HtmlContentFooBar->setSource()** method checks the user’s privileges for the page and then acquires the data to construct the page. The **$FormTable** in the method is an instance of an **HtmlFormTable**, which is simply a standard EAL table of data. Using the **FooBar.xml** configuration file, which defines the **FooBars** table, and then feeding that table the appropriate data using an Xpath query on the **SimpleXml** object, the *$FormTable* constructs the html source code for displaying a table of FooBar data. If the **HtmlFormTable->setEditPrivileges()** method is accessed with a value of “true”, then the **HtmlFormTable** will also build a form version which the user can edit, this will instantiate the **FormField** objects inside the form, such as date fields, text inputs, and drop-down selectors.

If the user has edit privileges, then an **HtmlForm** object is instantiated to contain the content of the form. The buttons (“Save”, “Delete”, “Reset”) for this form are defined in the page configuration file, as are any hidden fields that apply to the form as a whole, such as key data needed by the database.

With this strategy, the building blocks of a web page are put together by putting objects within objects in a logical fashion, illustrated in the following diagram:



Figure - Relationships between Html Objects.

**HtmlContent** objects can have any number of **Form** objects, which can have any number of block or table objects, which can have any number of **FormField** objects.

Open a browser window and navigate to the following URL:

*http://localhost/practice/eal/FooBar/*

The FooBars page should be displayed, clicking on the “Edit FooBars” link will toggle the page view to the edit mode. Below you will see how the elements of the page are related to their corresponding objects in PHP.



Figure - Html Object Interpreted by the Web Browser

Detecting changes to the form and [AJAX](http://www.w3schools.com/ajax/default.asp) calls to the server are included by the framework. So when you click “Save” on the form without any changes, an alert message will be presented. When modifications are made to the form and “Save” is clicked, the javascript files included by the framework on the page make an [XmlHttpRequest](http://www.w3schools.com/xml/xml_http.asp), posting the form content to the **/framework/api/processForm.php** file, which, based on the page code, routes the content to the appropriate form handler.

**/classes/practice/presentation/processform/FooBarProcessForm.php**

This is the class that receives the form content as an array, loops through it and determines what actions need to be taken based on the form values. The client sends both the original form values and the new form values, and based on these the **ProcessForm** class determines if the user wants to insert, update, or delete each record. In the case of our FooBars form, the process form checks if the “delete” field for the row is checked for delete, for an empty “id” hidden field for insert, and loops through the row fields, comparing them against their original value, to determine if an update needs to occur.

Uncomment the following line in the file, save, and attempt saving FooBar records in the web browser:

*ErrorCollection::addUserErrorMessage(0,"formValuesArray:".print\_r($this->formValuesArray));*

The **ErrorCollection::addUserErrorMessage()** static method gathers any errors that occurred in the processing of the form, such as missing entries, lack of changes to the data, or database errors, which are displayed to the user. The **ProcessForm** class checks if there are user error messages before attempting to send the delta list to the business layer.

In this case, we are returning the array of raw values submitted from the form. The fields prefaced with “original” are the form values captured when the screen first loaded, with the other fields being what was just submitted to the server.

Comment in the above line and uncomment the following line in the file, save, and attempt saving FooBar records in the web browser

*ErrorCollection::addUserErrorMessage(0, "deltaList:" . print\_r($deltaList));*

The data returned in the **DeltaList** are the list of **FooBarDelta** objects intended for transactions. The flags I, U, and D indicate what type of transaction is to occur.

If there are no user error messages in the **ErrorCollection**, then the **ProcessForm** will access the **FooBarBusinessComponent->saveFooBars()** method. If no exceptions are caught from this, the **ProcessForm** returns a success message; otherwise, it collects the exception and returns and error to the user.

**Building a Screen Sequence Diagram**

The following sequence diagram demonstrates the typical flow through the different objects within the framework in retrieving a screen to present the user. While our FooBar demonstration is very simple, it is possible to access several **HtmlContents**, **BusinessObjects**, **BusinessComponents**, and **DataAccessComponents**. Several objects, such as factories, utilities, and autoload, are not shown here for simplicity.

****

**Submitting a Form Sequence Diagram**

This sequence diagram illustrates the standard interactions between objects in the framework when performing a transaction, such as insert, update, or delete. Note that the Client submits an XmlHttpRequest, which comes from the form being submitted via an AJAX request. As with the Sequence for building a screen, multiple forms may be processed in a single transaction, and it is possible to access several **BusinessComponent** and **DataAccessComponent** objects to accomplish multiple transactions.

****

**Adding Functionality to Your Application**

Now we will walk through the process of making a minor change to our FooBar functionality. For our exercise, we will add a new “logical” property to our **FooBar** class, which will be a Boolean datatype.

Drop and reload the FooBar table to add a logical data type column like so, replacing “*username*” with your own name. The “logical” column will hold “Y” or “N” values:

DROP TABLE *username*\_foobar

;\p\g

CREATE TABLE *username*\_foobar

(

 name VARCHAR(25) NOT NULL WITH DEFAULT

 ,value VARCHAR(25) NOT NULL WITH DEFAULT

 ,logical CHAR(1) NOT NULL WITH DEFAULT

 ,date DATE NOT NULL WITH DEFAULT

)

;\p\g

grant all on table *username*\_foobar to public

;\p\g

COMMIT;\p\g

**Business Layer Modifications**

1. Begin by editing the **FooBar** class to include a protected property labeled *logical*. Validate the property in the constructor using the PHP [is\_bool()](http://php.net/manual/en/function.is-bool.php) function. Be sure to include a public **getLogical()** method that returns the value for *logical*.
2. Since the **FooBarDelta** refers to the FooBar constructor’s validation, it will also need to be modified to check *logical* properties submitted to it through its parent class.
3. The **FooBarBusinessObject** will need to be modified to include a node for logical values in the xml it generates.

**Persistence Layer Modifications**

1. Modify the **FooBarDataAccessComponent.php** file to include this new column in our SQL queries and in populating **FooBar** objects. Note that, because the database stores this flag as a “Y” or “N” in a CHAR(1) column, these values will need to be converted to *true* or *false* before they are added to the **FooBar** object. There are several ways to accomplish this, you could create a new protected function to convert the values or use a [single-line conditional statement](http://www.pixel2life.com/publish/tutorials/887/quickly_assign_variables_a_value_using_if_else_in_php/) to produce the same result. The correct solution would be to create a static utility class to encapsulate this conversion that all objects could access; however, that’s beyond the scope of this exercise.
2. At this point, you should be properly retrieving the data from the persistence layer and bringing it into the Business Objects for the page. You can verify this by inserting a few rows to the table at your discretion, uncommenting the *$pageObjectsXml->Foobars* echo statements in the **HtmlContentFooBar** class, and refreshing the page in your browser window. The additional properties will not appear in the table because we have yet to add them to the presentation layer.

**Presentation Layer Modifications**

1. Modify the **FooBar.xml** file, adding a new formfield node to the FooBars node with an appropriate label, id, and valueXpath to retrieve the *logical* value from the XML. Because we are dealing with a Boolean value, a *formfield->type* of “*Checkbox*” is most appropriate. This should add a new column to your table displaying the logical value for the FooBar row.

**Note:** Refreshing the page in the browser window will likely not update the form display. This is because the page configuration is stored in the application session. In order to see the additional table column, you will need to close out the browser and relaunch the page to see the change take effect.

**Debugging the Application**

In the web browser, attempt to edit the FooBar Records and save. This should return an error. Now we will walk through the various strategies for debugging it.

1. **Review the Application Log Files:** in the server root directory, access the “Log” directory and locate the log file with today’s date: *YYYYMMDD*ERROR.log (ie. “201004ERROR.log”). Open this file in your preferred text editor, such as UltraEdit or your IDE.

You will find an entry similar to the following:

1. [2010/04/22 10:11:11]
2. /almis/framework/api/processForm.php
3. ealdeveloper
4. [-----PAGE ERRORS-----]
5. errorId [76c1c4f8b80564f6bd8ef3d60a7b0881]
6. type [ERROR]
7. exceptionClass [BusinessExceptionInvalidArgument]
8. errno [0]
9. errstr [Attempt to add invalid logical flag to Test FooBar.]
10. errfile [C:\Program Files\Apache Software Foundation\Apache2.2\htdocs\almis\application\eal\classes\practice\business\entity\FooBar.php]
11. errline [87]
12. errcontext [#0 C:\Program Files\Apache Software Foundation\Apache2.2\htdocs\almis\application\eal\classes\practice\business\entity\FooBarDelta.php(49): FooBar->\_\_construct('', 'Test', 'Testing', '04/22/2010', 'I')
13. #1 C:\Program Files\Apache Software Foundation\Apache2.2\htdocs\almis\application\eal\classes\practice\presentation\processform\FooBarProcessForm.php(99): FooBarDelta->\_\_construct('', 'Test', 'Testing', '04/22/2010', 'I')
14. #2 C:\Program Files\Apache Software Foundation\Apache2.2\htdocs\almis\framework\api\processForm.php(67): FooBarProcessForm->process()
15. #3 {main}]
16. datetime [04/22/2010 10:11:11]
* **Line 1 and 16** are the date and time the exception occurred
* **line 2** is the parent file in the application where the exception occurred
* **line 3** is the logged in user
* **line 7** is the type of exception that was thrown
* **line 9** is the user-friendly message for the exception
* **lines 10 through 15** are the trace stack for the application, outlining the sequence of files and class methods accessed that led up to the exception with the most recent first. From the above example in **lines 10 and 11**, we can see that the exception was thrown from the **FooBar** object at line 87 in **FooBar.php**, which was accessed by the **FooBarDelta** at line 49, which was accessed by the **FooBarProcessForm** at line 99.
1. **IDE Debug Method (Netbeans):** open the **FooBarProcessForm** in your IDE and set a debugging breakpoint around line 87 and 93, where a new **FooBarDelta** is being added to the **$deltaList[]** by clicking on the line number in the IDE beside the code so that a pink square appears there and the line is highlighted pink:



Edit the **index.php** file in the **practice/** directory to load “FooBar” as the default page if the server is “localhost” around lines 118 and 120.

Run debug in Netbeans:



The debugger will automatically break at the beginning of the application, click the *Continue* or *F5* to run the application to the next break point:



A browser window should load the “**FooBars**” page. Edit the form and save it. In the IDE, the debugger will now have stopped at the beginning of the **processForm.php** file. Click *Continue* or *F5*. The application should run to first breakpoint. Use the **Step Into (F7)** option in the debugging console, use **Step Over (F8)** in the **autoload** class to skip this standard functionality.



When you step into the **FooBar** class from the **FooBarDelta** class, select **Window>Debugging>Variables** to bring up a list of the variables being used in the current context. The issue should be apparent:



1. Not covered in the exercise is the use of echo statements for debugging. These work similar to uncommenting the debugging lines of code in strategic points in the application using the [print\_r()](http://php.net/manual/en/function.print-r.php) or [var\_dump()](http://php.net/manual/en/function.var-dump.php) PHP functions to see the details of an object or array, and the [echo](http://php.net/manual/en/function.echo.php) statement to view the values of specific variables. The use of debugging statements in code is discouraged due to the possibility of leaving such statements in the code, and accidentally having them make it into production.
2. Edit the **FooBarProcessForm** to include the *logical* property when generating the **FooBarDelta**. Your FooBar Records screen should now CRUD FooBar records properly.

**Further Study**

1. Run Debugger in NetBeans and try stepping through other pages in the application, and through code not covered in this document. If you have questions, please feel free to contact a Framework developer.
2. Once completed with your exploring and experimenting with the framework, be sure to drop the *username*\_foobar table from the database and set **almis** to your main project in NetBeans.