4
Interfacing with SugarCRM

Hopefully, you are feeling very confident about customizing SugarCRM. Therefore, this seems an appropriate point to take a step back from the customization process, and have a deeper look into the structure of SugarCRM itself. So, what we'll do now is:

- See how the SugarCRM application is put together as we examine the user and data interfaces in this chapter.
- In the next chapter we'll see how the SugarCRM database is put together.

What Have we Learned so Far?

Over the past three chapters we've actually learned quite a bit about the application architecture. To start with:

- The application consists of a number of PHP files on a web server.
- The application requires a database in the background.

So, if we think about the PHP files we know that:

- We always access the SugarCRM application via a central PHP file—`index.php`.
- We have the `custom` directory for storing any language customizations.
- We have a `themes` directory where we store the files for customizing colors, fonts, icons, and images for the application.
- SugarCRM consists of a number of `module` directories, which provide the actual SugarCRM functionality. They're all stored in the `modules` directory.

Let us just remind ourselves about the files that each of these directories needs to contain.

The Include Directory

The `include` directory contains module-independent files such as:

- `modules.php`

The Custom Directory

The `custom` directory contains:

- `custom/include/language/en_us.lang.php`
- `custom/modules/<module>/language/en_us.lang.php`

The Themes Directory

The `themes` directory contains a directory for each theme to be used by your application. Each of these directories must have:

- `config.php`
- `style.css`

The Modules Directory

The `modules` directory contains a directory for each module to be used by your application. Each of these directories *must* have:

- `index.php`
- `Forms.php`
- `<module name>.php`
- `language/en_us.lang.php`

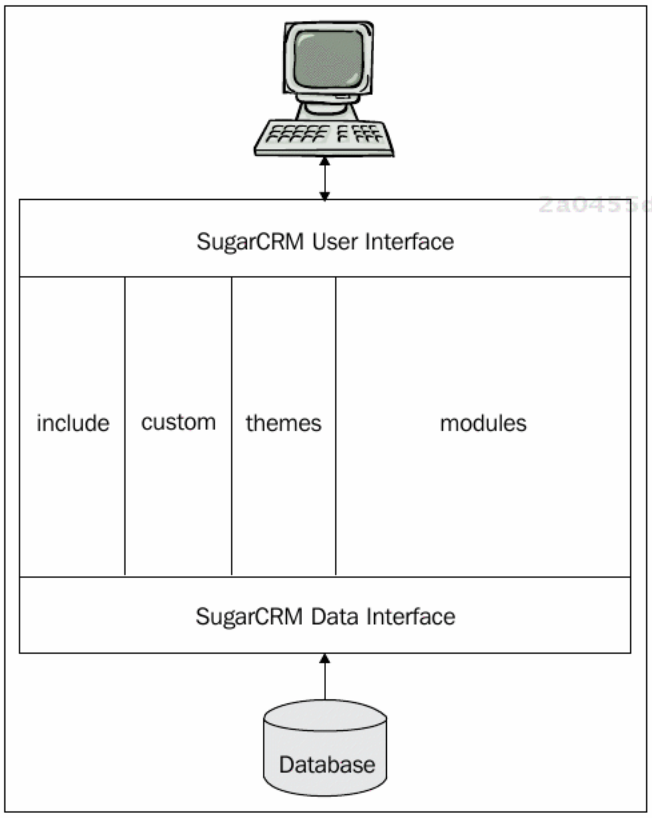
We've also learned that the SugarCRM modules have an interface to the database, and in particular:

- Custom fields can be defined on the database, but the application caches details about them in `cache/dynamic_fields`.
- Each module has its own data field definitions in a file named `vardefs.php`.

From all of this we can already build ourselves a general picture of the SugarCRM application architecture.

Overview of the SugarCRM Application Architecture

The SugarCRM application architecture is simple, but effective:



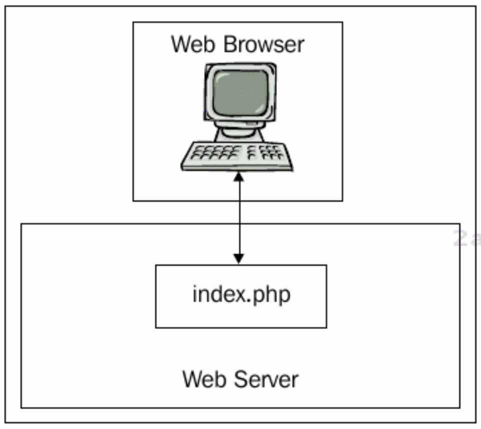
As you can see form the diagram above, and, as you may well have worked out for yourself already from the last three chapters, our users use their computers (i.e. their web browsers) to access the Sugar User Interface – this then governs all interactions between the user and the SugarCRM functionality.

You will also see that there is another interface (the SugarCRM data interface) between the SugarCRM functionality and the underlying database (as you would expect).

In the remainder of the chapter we'll concentrate on the user and data interfaces, and then in Chapter 5 we'll look at the database itself.

The SugarCRM User Interface

You probably have worked out that the SugarCRM user interface is actually generated by the `index.php` file in your main SugarCRM directory:



So, there's nothing here that you don't already know. It is, therefore, worth having a look at what the interface actually does for us.

The user interface (or if you prefer – the UI layer):

- Decodes information posted (via the HTML forms) to the SugarCRM forms
- Authenticates users' log on details and active sessions
- Provides a wrapper around the modules files

In other words all a user has to do is to call up `index.php` on the web server, and it will do all of the work for them.

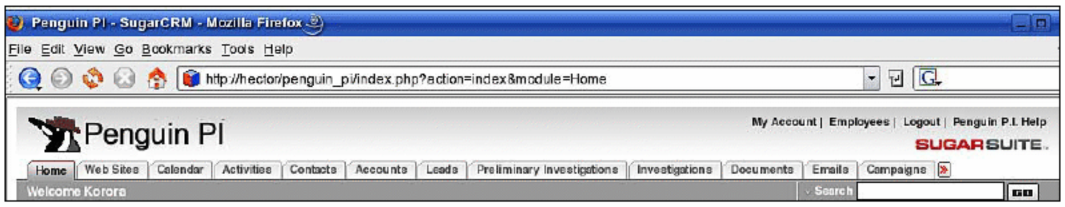
Calling Modules

Having identified that `index.php` handles all of our interactions with SugarCRM, it's worth just looking at how we can use the user interface to guide us to particular modules, and, perhaps more importantly, how we can use it to carry out actions.

There are two key parameters that you can pass to `index.php`:

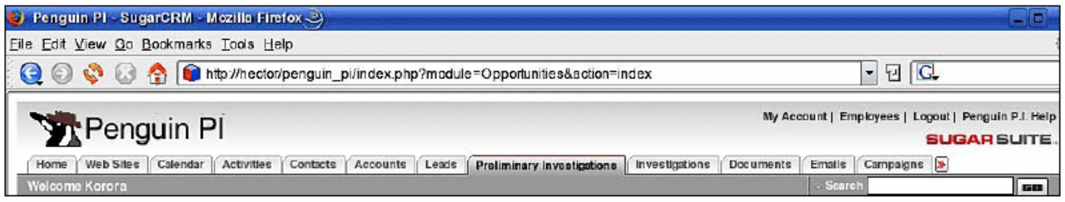
- **module** – This, obviously, is the module that you want to call. However, to be completely correct, it is the *directory* in which the module is stored.
- **action** – This is the PHP file in the module directory to be used. By default its `index.php` (i.e. the `index.php` file in the module directory, not `index.php` in the top level of the web server). However, you can call other PHP files in the directory.

So, let us imagine Korora logging on – she'll start by typing in the SugarCRM URL (in her case `http://hector/penguin_pi`) however, once she's finished typing in her user name and password then she'll see:

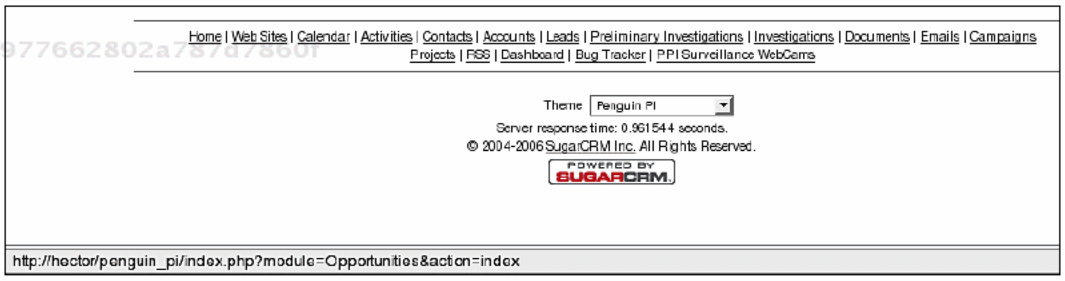

Penguin PI - SugarCRM - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://hector/penguin_pi/index.php?action=index&module=Home
Penguin PI
My Account | Employees | Logout | Penguin P.I. Help
SUGARSUITE
Home | Web Sites | Calendar | Activities | Contacts | Accounts | Leads | Preliminary Investigations | Investigations | Documents | Emails | Campaigns
Welcome Korora
Search
GO

As you can see the user interface has set the module to **Home**, and the action to **index**.

If Korora then clicks on one of the tabs (for example **Preliminary Investigations**) then the user interface handles this change for her, and you can see that this has been done by setting the module to **Opportunities** and the action to **index**:


Penguin PI - SugarCRM - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://hector/penguin_pi/index.php?module=Opportunities&action=index
Penguin PI
My Account | Employees | Logout | Penguin P.I. Help
SUGARSUITE
Home | Web Sites | Calendar | Activities | Contacts | Accounts | Leads | Preliminary Investigations | Investigations | Documents | Emails | Campaigns
Welcome Korora
Search
GO

In fact, you'll find that each of the tab titles is actually a link, and each of the links simply passes the appropriate module and index back to the main `index.php`. For example, if you place the mouse pointer over the **Preliminary Investigations** tab title then you'll see that the link address is:


Home | Web Sites | Calendar | Activities | Contacts | Accounts | Leads | Preliminary Investigations | Investigations | Documents | Emails | Campaigns
Projects | RSS | Dashboard | Bug Tracker | PPI Surveillance WebCams
Theme Penguin PI
POWERED BY SUGARCRM

Server response time: 0.961544 seconds.
© 2004-2006 SugarCRM Inc. All Rights Reserved.

http://hector/penguin_pi/index.php?module=Opportunities&action=index

We can now use this knowledge to manage the way in which we use SugarCRM. For example, if we return to the module tab that we created in Chapter 2, then we can change it so that it contains a list of key 'jobs' to be done.

We could start by changing the title of the module by going to `custom/include/language`, editing `en_us.lang.php`, and changing:

```
'TestApp' => 'PPI Surveillance WebCams',
```

to:

```
'TestApp' => 'Daily Tasks',
```

Next we can think about editing `modules/TestApp/index.php` so that Korora's daily tasks are displayed. And, to make it even more useful, we can make use of the `strftime` function (which formats the local time) to display different tasks at different times:

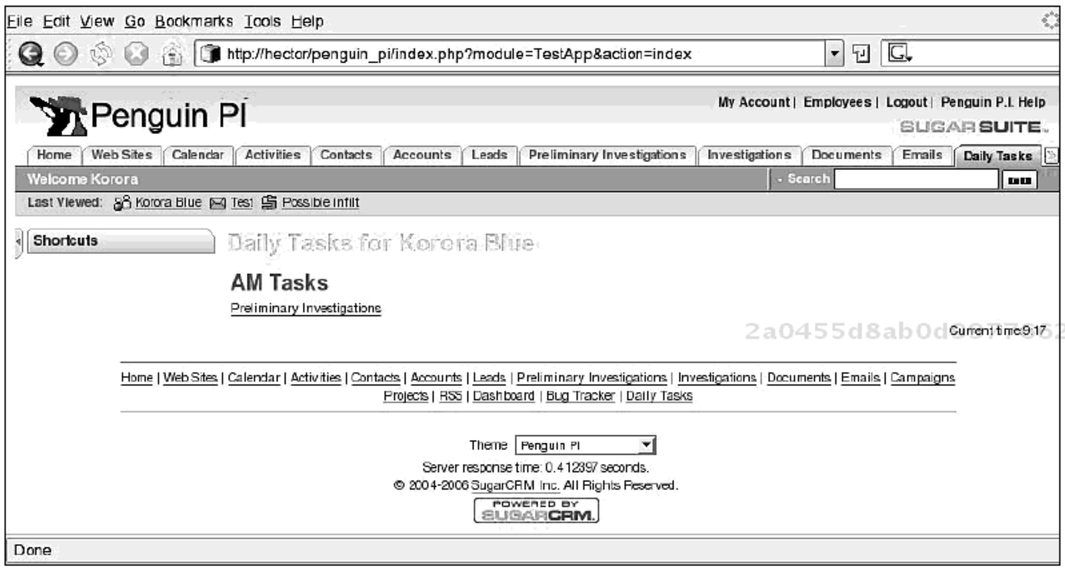
```
<?php
global $current_user;
#Get the local time (from the server)
$h = strftime("%H");
$m = strftime("%M");
?>
<h1>Daily Tasks for
<?php
#Display the users name (to be more personal just use the first name)
echo $current_user->first_name . " " . $current_user->last_name;
?></h1>

<table width=100%>

<?php
#Display tasks for the morning
if ( $h >= 9) { ?>
<tr><td><h2>AM Tasks</h2></td></tr>
<tr><td>
<a href=index.php?module=Opportunities&action=index>
Preliminary Investigations
</a></td></tr>
<?php } ?>

<?php
#Display tasks for the afternoon
if ( $h >= 12) { ?>
<tr><td><hr></td></tr>
<tr><td><h2>PM Tasks</h2></td></tr>
<tr><td>
<a href=index.php?module=Cases&action=index>
Investigations
</a></td></tr>
<?php } ?>
<tr><td align=right>
<?php
#And finally show the current (server) time
echo "Current time:" . $h . ":" . $m; ?>
</td></tr>
</table>
```

The end result (in the morning) is:


File Edit View Go Bookmarks Tools Help
http://hector/penguin_pi/index.php?module=TestApp&action=index
Penguin PI
My Account | Employees | Logout | Penguin P.I. Help
SUGARSUITE
Home Web Sites Calendar Activities Contacts Accounts Leads Preliminary Investigations Investigations Documents Emails Daily Tasks
Welcome Korora
Search
Last Viewed: Korora Blue Test Possible infilt
Shortcuts

Daily Tasks for Korora Blue

AM Tasks

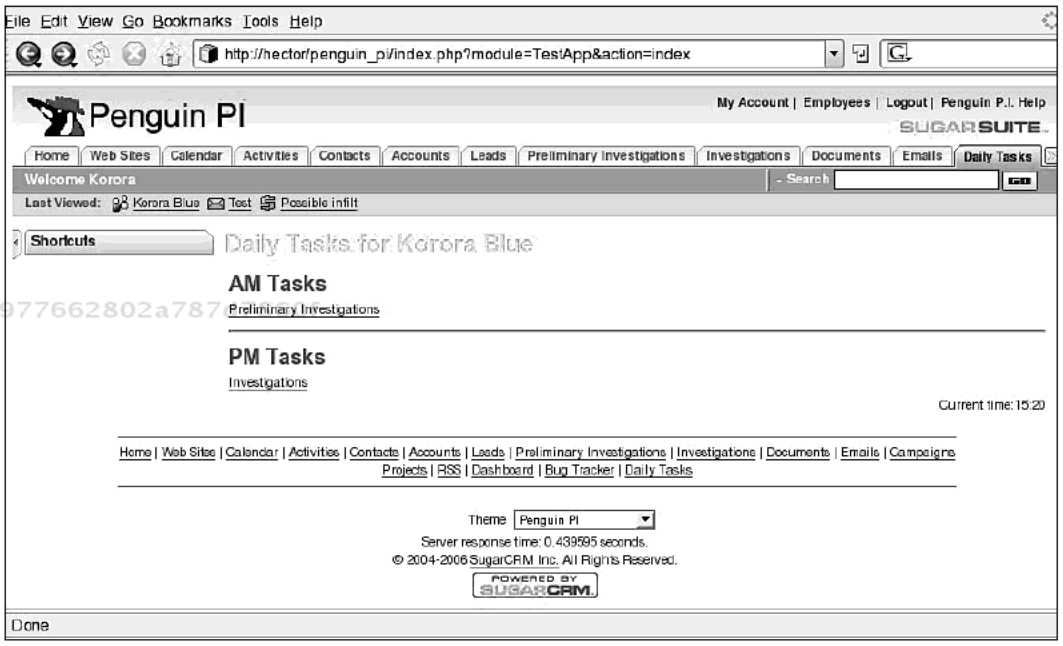
Preliminary Investigations

Current time:9:17

Home | Web Sites | Calendar | Activities | Contacts | Accounts | Leads | Preliminary Investigations | Investigations | Documents | Emails | Campaigns
Projects | RSS | Dashboard | Bug Tracker | Daily Tasks
Theme Penguin PI

Server response time: 0.412397 seconds.
© 2004-2006 SugarCRM Inc. All Rights Reserved.

And in the afternoon:


File Edit View Go Bookmarks Tools Help
http://hector/penguin_pi/index.php?module=TestApp&action=index
Penguin PI
My Account | Employees | Logout | Penguin P.I. Help
SUGARSUITE
Home Web Sites Calendar Activities Contacts Accounts Leads Preliminary Investigations Investigations Documents Emails Daily Tasks
Welcome Korora
Search
Last Viewed: Korora Blue Test Possible infilt
Shortcuts

Daily Tasks for Korora Blue

AM Tasks

Preliminary Investigations

PM Tasks

Investigations

Current time: 15:20

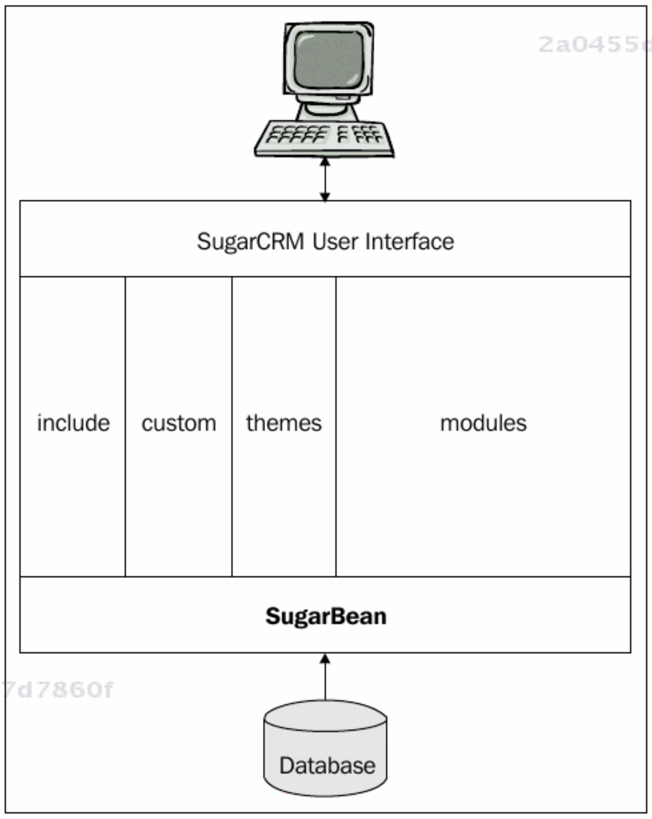
Home | Web Sites | Calendar | Activities | Contacts | Accounts | Leads | Preliminary Investigations | Investigations | Documents | Emails | Campaigns
Projects | RSS | Dashboard | Bug Tracker | Daily Tasks
Theme Penguin PI

Server response time: 0.439595 seconds.
© 2004-2006 SugarCRM Inc. All Rights Reserved.

Now that we've had a look at the SugarCRM user interface it's time to move on to the SugarCRM Data Interface—otherwise known as SugarBean.

SugarBean—The SugarCRM Data Interface

We'll be looking at the structure of the database in Chapter 5, but it's possible that you will never have to access it, and that's because of SugarCRM's SugarBean:



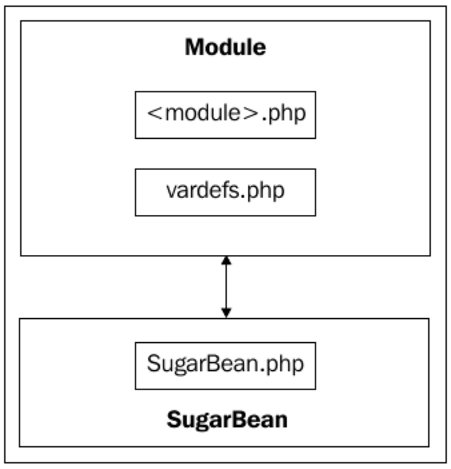
So, what is the SugarBean? At its simplest level it's another PHP file, but it does a very important job—it's a high-level API that allows you to manipulate your business data without having to worry (too much) about what's going on in the database. The SugarBean:

- Is the base class for the entire SugarCRM business object that you need to use. This means that the **Opportunity** object (for example) is just an extension of the SugarBean.
- It supplies all of the key functions for your business objects, such as creating records, retrieving records, updating and deleting.

And, as you would expect, the SugarBean consists of a set of PHP files.

The SugarBean Files

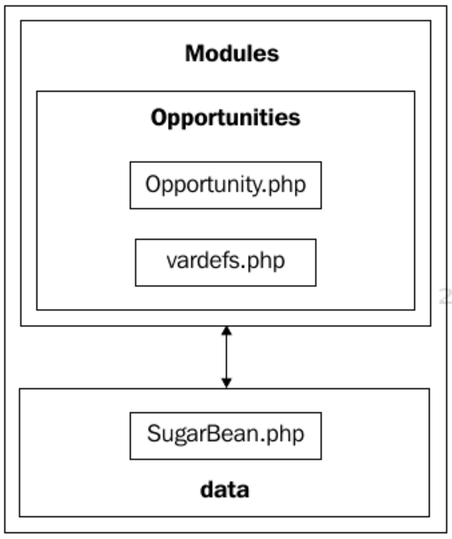
As we've already learned the SugarBean is the data interface between our modules and our database, and it consists of a number of PHP files on the SugarCRM web server:



You can see that there are three key files for the SugarBean:

- `SugarBean.php`—This is located in the `data` directory and is (as we've already learned) the base class file.
- `vardefs.php`—This is the schema for the business object. There is one for each module that uses the SugarBean.
- `<module>.php`—Each module using the SugarBean must contain this file, and it is used to extend the base class for the particular module. It is not actually named the same as the module, but takes the singular form, e.g. the **Opportunities** module would contain `Opportunity.php`.

So, if we continue to think about Opportunities for a moment then we'd see the following set up:



In order to better understand the SugarBean let's start by examining `vardefs.php` in a little bit more detail.

vardefs.php

You'll hopefully remember that we have already worked with the `vardefs.php` file. In Chapter 3 we saw that it is possible to add SugarCRM fields into the mass update sub-screen by editing this file – we modfied `modules/Emails/vardefs.php`, and updated the `massupdate` property:

```
'date_start' => array (
                 'name' => 'date_start',
                 'vname' => 'LBL_DATE',
                 'type' => 'date',
                 'len' => '255',
                 'rel_field' => 'time_start',
                 'massupdate'=>true,
                 'comment' => 'Date of last inbound email check', ),
```

Last time we just made the changes and moved on, but this time we'll look each of the properties, although now that you know that this is the database schema file, I'm sure that you can work out most of the details yourself.

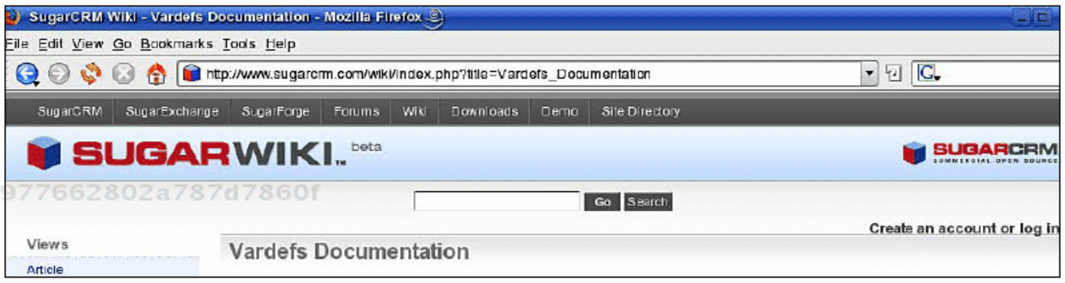
In case you haven't worked out what's going on here – this array represents a single field in the database schema, and you'll see that each field has a set of parameters. In this case `'date_start'` has:

- `'name'` – Unsurprisingly this is the name of the field.
- `'vname'` – The field label ID for the module's `en_us.lang.php` file.
- `'type'` – The data type of the property.
- `'len'` – The length of the field.
- `'rel_field'` – Since this is a date field it has a related time field.
- `'massupdate'` – You already know what this does (but in case you've forgotten – you set this if you want to be able to update a group of records all at the same time).
- `'comment'` – That would be a comment then.

There are actually a lot more parameters that are available to you, but this is still a fairly fluid area of SugarCRM, and these are liable to change. For that reason I'm not going to give you an exhaustive list. Instead it's time to look at some of SugarCRM's on-line documentation.

vardefs On-line Documentation

- You'll find current details about `vardefs` at `http://www.sugarcrm.com/wiki/index.php?title=Vardefs_Documentation`:


SugarCRM Wiki - Vardefs Documentation - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://www.sugarcrm.com/wiki/index.php?title=Vardefs_Documentation
SugarCRM SugarExchange SugarForge Forums Wiki Downloads Demo Site Directory
SUGARWIKI beta
SUGARCRM
Go Search
Create an account or log in
Views
Article
Vardefs Documentation

Having just said that you should refer to the on-line SugarWiki to obtain an up to date list of all of the available parameters for the vardef fields, it is still worth looking at one parameter – the type.

vardefs Field Types

There are a number of different field types available to you for use in the data schema—some of which you'll recognize if you've worked databases, and some of which are specific to SugarCRM:

- `'assigned_user_name'`—Contains a SugarCRM user name
- `'blob'`—the **B**inary **L**arge **OB**ject—Normally used when you want to store a large amount of data in a single field
- `'bool'`—A boolean value, although it uses a 1 or 0 rather than true or false and in fact it maps to tiny integer on a MySQL database
- `'char'`—An array of characters—although you'd never use this, since varchar is available
- `currency`
- `'date'`
- `'datetime'`
- `'email'`
- `'enum'`—Enumeration—normally used for dropdown lists
- `'float'`—A decimal number—normally used to store currency
- `'id'`—A 36 character SugarCRM ID number
- `'int'`—Integer
- `'link'`—A relationship link
- `'nondb'`—A derived value—not from database (and not technically a type), which could come from a PHP function
- `'num'`—Interesting one—this is actually stored in the MySQL database as a varchar
- `'phone'`—a phone number
- `'relate'`—Related Bean, i.e. related to a field in another table
- `'text'`—text field. Basically a 'char' that holds 65,535 characters
- `'varchar'`—A variable sized string, the length of which is set by the 'len' field

So, nothing really contentious here—the list contains all the common field types that you will need for your project.

The Complete vardefs File

So far we've only looked at an individual field within `vardefs.php`; however, that's not the end of the story. Each field is defined as an array of parameters, but these are just part of a larger array of fields, and are stored in another array—the dictionary:

```
$dictionary['Opportunity'] = array(
  'table' => 'opportunities',
  'audited'=>true,
  'unified_search' => true,
  'duplicate_merge'=>true,
  'comment' => 'An opportunity is the target of selling activities',
  'fields' => array )
```

We're now going to leave `vardefs.php` again, but we will be coming back to it—in Chapter 8, when we'll be looking at developing a complete module. In the meantime we'll have a quick look at the `<module>.php` file.

The <module>.php File

Although I've referred to the `<module>.php` file don't forget that the file must actually be given the singular name for the module, so for Opportunities use `Opportunity.php`, for Emails use `Email.php`, and so on.

Our file contains a class that extends the basic SugarBean class (`SugarBean.php`), and it's used to define:

- Variables mapped to the database schema (`vardefs.php`)
- Any additional functionality specific to the particular business object

The class files all have the same format, and so if we look at `Opportunity.php`, we'll see that the base class file is loaded, along with any others that are required:

```
require_once('data/SugarBean.php');
require_once('modules/Contacts/Contact.php');
require_once('modules/Tasks/Task.php');
require_once('modules/Notes/Note.php');
require_once('modules/Calls/Call.php');
require_once('modules/Leads/Lead.php');
require_once('modules/Emails/Email.php');
require_once('include/utils.php');
```

And then the new class is defined:

```
class Opportunity extends SugarBean
{
  var $field_name_map;
  // Stored fields
  var $id;
  var $lead_source;
  var $date_entered;
  var $date_modified;
  var $modified_user_id;
}
```

And, of course, it will need a constructor function:

```
function Opportunity()
{
   parent::SugarBean();
   global $sugar_config;
   if(!$sugar_config['require_accounts'])
   {
      unset($this->required_fields['account_name']);
   }
   global $current_user;
 }
```

As well as any extra functions required for the Opportunity class:

```
function get_list_view_data()
{
  global $locale, $current_language, $current_user, $mod_strings,
  $app_list_strings, $sugar_config;
  $app_strings = return_application_language($current_language);
  require_once('modules/Currencies/Currency.php');
  $temp_array = $this->get_list_view_array();
  #Set the sales state
  $temp_array['SALES_STAGE'] =
    empty($temp_array['SALES_STAGE']) ? '' :
                                    $temp_array['SALES_STAGE'];
  #Set the ammount
  $temp_array['AMOUNT'] = currency_format_number($this->amount);
  #Set the name
  $temp_array["ENCODED_NAME"]=$this->name;
  #Return the result
  return $temp_array;
}
```

Now, just like `vardefs.php`, we're going to leave the class file behind for now, and return to it in Chapter 8. However, you, no doubt, want to see the SugarBean in action—so we'll turn our attention to SugarCRM's logic hooks.

SugarBean in Action—SugarCRM's Logic Hooks

You may not have heard of logic hooks before—if not then, quite simply, they provide us with the ability to add in our own custom business logic into the SugarCRM applications. These logic hooks may take the form of some kind of validation, or they may take the form of a more involved business operation.

If we look at the Penguin P.I. office for a moment, we might see Korora sat at her desk. One of her tasks is to evaluate any new preliminary investigations and then assign them to someone. However, when she does this she must ensure that:

- Only people with certain roles can receive preliminary investigations.
- Each preliminary investigation must got to the correct office covering the geographical region in which the investigation is to be carried out.
- Where more than one person qualifies for receiving the preliminary investigation then the person with the least amount of work must be chosen.

We've therefore got two options:

- Let Korora work it all out for herself—regardless of how long it's going to take.
- Add a logic hook that will do all of this automatically.

We're not going to do all of that at the moment; we'll save that for Chapter 9 when we'll look at developing custom workflows within SugarCRM. However, what we will do is create a logic hook that records changes in assigned users for any Opportunity.

Now, if you're already used to working with databases such as Oracle then you'll be used to the concept of a *trigger*. Triggers are simply pieces of code that are run when particular events (such as update or insert) occur on the database—and that's exactly what the logic hook does—it runs a PHP file when the SugarBean carries out certain database operations. These key events are:

- `after_retrieve`
- `before_save`
- `before_delete`
- `after_delete`
- `before_undelete`
- `after_undelete`

So, in this case we want the logic hook to operate on the before_save event. We're also going to be writing to a log file in this instance, and so the first thing to do is to write the code for that. We want to keep this separate from the standard SugarCRM code and so we'll place it custom/include and call the file penguin_pi_functions.php:

```
#File:  penguin_pi_functions.php
<?php
function WriteToLogFile($strText) {
  $File = '/www/penguin_pi/test.log'; #Choose a suitable file name
  $Handle = fopen($File, 'a'); #Open the file
  $Data = $strText . "\n"; #Add a carriage return to the text
  if($Handle) { // avoid further errors on file access failure
  fwrite($Handle, $Data); #Write the text to the file
  fclose($Handle); #Close the file
  }
}
?>
```

And just a note – you may need to manually create (and set the permissions for) the log file before you run the code.

Next we'll need the code file that's going to be run by the logic hook. Again, we'll put it in custom/include, but this time we'll call the file ppi_prelim_change.php, and it's another class file:

```
#File:  ppi_prelim_change.php
<?php
require_once('data/SugarBean.php');
require_once('modules/Opportunities/Opportunity.php');
require_once('custom/include/penguin_pi_functions.php');

class ppi_prelim_change {
  function ppi_prelim_change (&$bean, $event, $arguments) {
    global $sugar_config;

    if ($bean->fetched_row['assigned_user_id']!=
                                           $bean->assigned_user_id)
    {
      #Obtain the information for the old user
      $old_user = new User(); #Create a user object
      $old_user->retrieve($bean->fetched_row['assigned_user_id']);
      $old_assigned_user_name =
      $old_user->first_name.' '.$old_user->last_name;

      #Obtain the information for the new user
      $new_user = new User();
      $new_user->retrieve($bean->assigned_user_id);
      $new_assigned_user_name =
        $new_user->first_name.' '.$new_user->last_name;
        #Write the information to the log file
        WriteToLogFile
          ($old_assigned_user_name . " -> " . $new_assigned_user_name );
      }
    }
  }
?>
```

You'll notice from the code that both the current (i.e. changed) data and the original data are available to the function by making use of the *$bean* object:

- `$bean->assigned_user_id` provides the new user ID
- `$bean->fetched_row['assigned_user_id']` provides the old user ID.

You'll also notice that the code makes use of:

- The SugarBean base class file (`SugarBean.php`)
- The Opportunity class file (`Opportunity.php`)
- Our own custom functions file (`penguin_pi_functions.php`)

One very useful function worth taking note of is *retrieve*—you'll see from the code that this obtains the assigned user details with the minimum of effort on your part.

Finally, we just need to create the logic hook file itself. However, unlike the last two files, this *must* be placed in a specific location. You might expect that the Opportunities logic hook should be placed in `modules/Opportunities`, and you'd be nearly correct—it actually needs to be be placed in `custom/modules/Opportunities`, and it also has to be named `logic_hooks.php`:

```
#File: logic_hooks.php
<?php
if(!defined('sugarEntry') || !sugarEntry) die('Not A Valid Entry
Point');
$hook_array = Array(); #Create an array

$hook_array['before_save'] = Array(); #Create a sub-array

#Write the required information to the array
$hook_array['before_save'][] = Array(1, 'ppi_prelim_change',
  'custom/include/ppi_prelim_change.php',
  'ppi_prelim_change', 'ppi_prelim_change');
?>
```

If you examine the code then you'll see that we have to define an array (called `$hook_array`). This array then contains a sub-array, and it's this array that defines the logic hook itself.

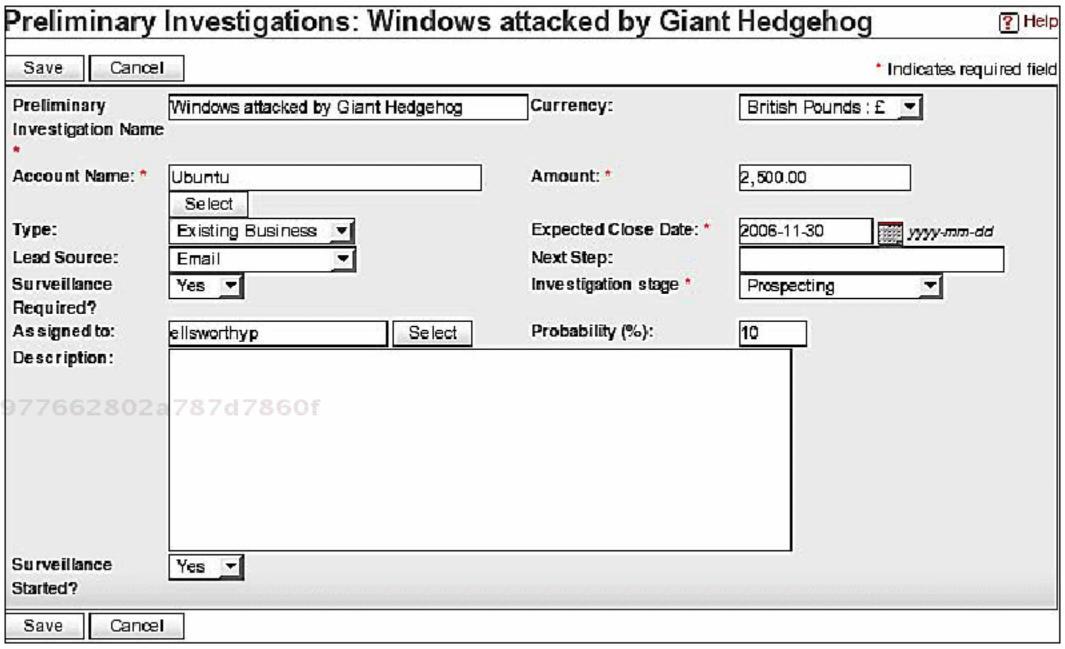
You'll notice that the array has the same name as the event on which the trigger is to be set, and it has a number of elements:

- Logic hook order – we can define a number of hooks in the same file, and this element defines the order in which they should be used.
- Name – this is just a placeholder to store the name of the hook.
- PHP code file location.
- PHP class to be called.
- PHP function that is to be run by the hook.

If you have these three files in place then your logic hook is up and running, and just waiting for your users to do something.

The End Result of Using the Logic Hook

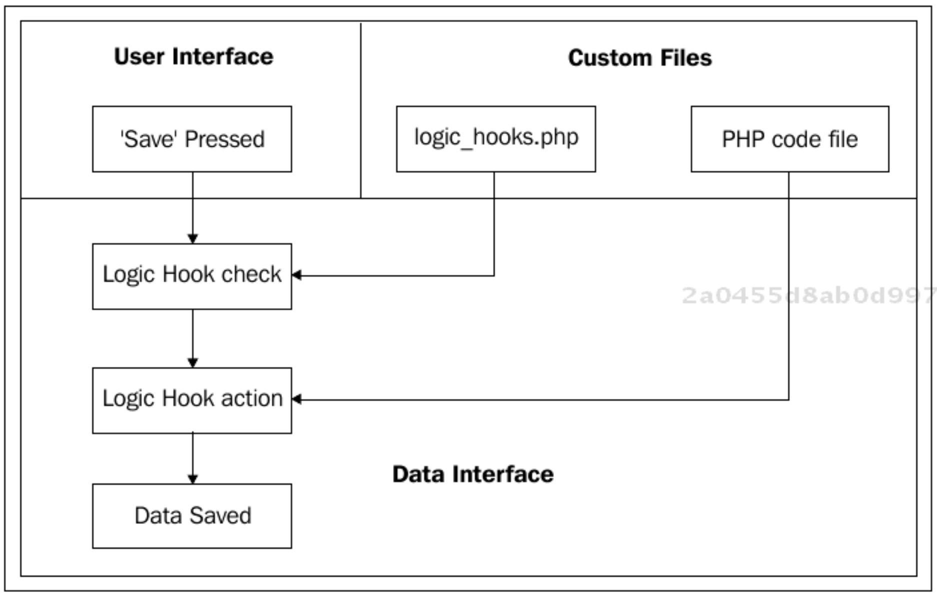
Back to Korora – she now needs to edit one of the preliminary investigations and change the assigned user:



Preliminary Investigations: Windows attacked by Giant Hedgehog

Preliminary Investigation Name *	Windows attacked by Giant Hedgehog	Currency:	British Pounds : £
Account Name: *	Ubuntu	Amount: *	2,500.00
Type:	Existing Business	Expected Close Date: *	2006-11-30 yyyy-mm-dd
Lead Source:	Email	Next Step:	
Surveillance Required?	Yes	Investigation stage *	Prospecting
Assigned to:	ellsworthyp	Probability (%):	10
Description:			
Surveillance Started?	Yes		

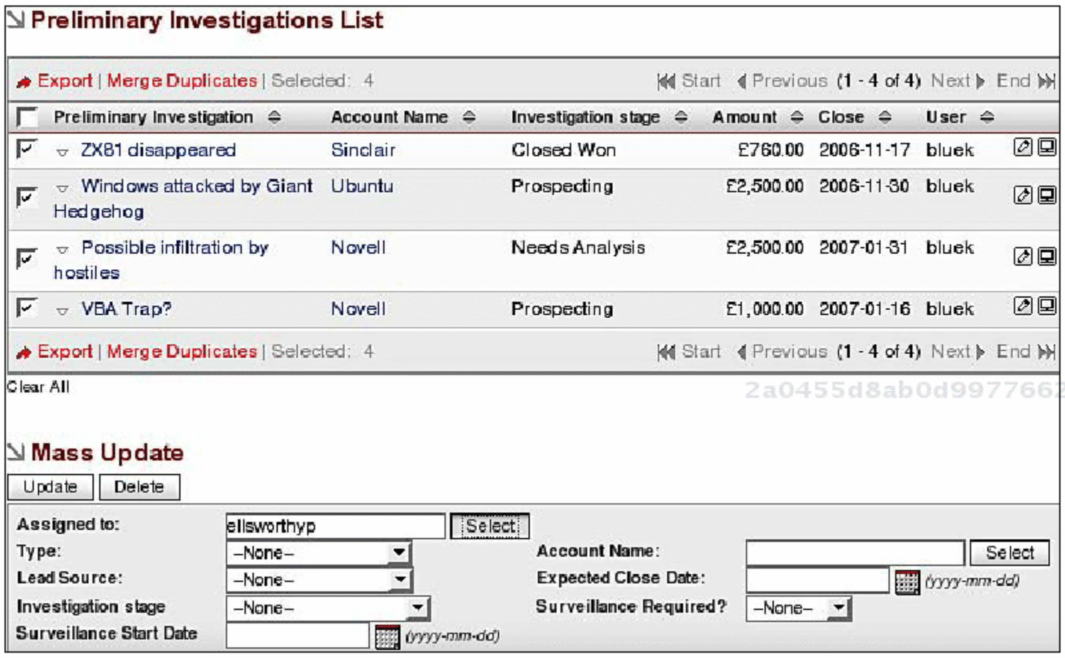
When she clicks the **Save** button then she'll be unaware of any differences in SugarCRM; however, in the background something will have changed:



Although your users will see nothing, SugarCRM will check to see if a Logic Hook exists. If it does then the associated PHP code file will be run, and (in this case) the data will be saved. And, of course, if Korora was to look on the web server she'd find that the `www/penguin_pi/test.log` file would contain a new entry:

```
Korora Blue -> Pygoscelis Ellsworthy
```

And this doesn't only work for individual instances—this will also work for the mass update. So, if you look back on the tab screen and actually carry out a mass update:



Preliminary Investigations List

Export | Merge Duplicates | Selected: 4 — Start Previous (1 - 4 of 4) Next End

	Preliminary Investigation	Account Name	Investigation stage	Amount	Close	User
✓	ZX81 disappeared	Sinclair	Closed Won	£760.00	2006-11-17	bluek
✓	Windows attacked by Giant Hedgehog	Ubuntu	Prospecting	£2,500.00	2006-11-30	bluek
✓	Possible infiltration by hostiles	Novell	Needs Analysis	£2,500.00	2007-01-31	bluek
✓	VBA Trap?	Novell	Prospecting	£1,000.00	2007-01-16	bluek

Export | Merge Duplicates | Selected: 4 — Start Previous (1 - 4 of 4) Next End

Clear All

Mass Update

Update Delete

Assigned to:	ellsworthyp		
Type:	–None–	Account Name:	
Lead Source:	–None–	Expected Close Date:	(yyyy-mm-dd)
Investigation stage	–None–	Surveillance Required?	–None–
Surveillance Start Date	(yyyy-mm-dd)		

You'll find that the logic hook fires for each record that you update.

Obviously this has been a very simple example, but we'll look at this more extensively in Chapter 9, and then we'll see how to use logic hooks as part of a workflow system.

Summary

In this chapter we've spent some time looking at the SugarCRM user interface and the data interface. We've seen how to use these effectively within our SugarCRM customizations.

The SugarBean is SugarCRM's high-level API that handles all our interactions with the database.

Logic hooks enable us to add in our own business logic into SugarCRM with the minimum effort. They are similar to database triggers – except that it's the SugarBean that does all the work and not the database.

Thus we've looked at the interfaces, and how to use them effectively in our customizations. We'll look at the files in more detail in Chapter 8 (when we'll develop a complete module), and Chapter 9 (when we'll look at custom workflows). However, before we do all that we'll examine the structure of the database itself.