
Sugar Developer Guide 6.1

- [Sugar Developer Guide 6.1](#)
 - [Preface](#)
 - [Introduction](#)
 - [Application Framework](#)
 - [Module Framework](#)
 - [Customizing Sugar](#)
 - [Sugar Logic](#)

Sugar Developer Guide 6.1

This page was not added to the PDF due to the following tag(s): article:topic-guide



Preface

1. [Overview](#)
 2. [The Sugar application](#)
 3. [The Sugar community](#)
 4. [Audience](#)
 5. [Feature Overview](#)
 6. [Core Features](#)
 - 6.1. [Sales Force Automation](#)
 - 6.2. [Marketing Automation](#)
 - 6.3. [Customer Support](#)
 - 6.4. [Collaboration](#)
 - 6.5. [Reporting](#)
 - 6.6. [Administration](#)
 7. [Related Documentation](#)
-

Overview

Welcome to Sugar, an open source Customer Relationship Management (CRM) application. Sugar enables organizations to efficiently organize, populate, and maintain information on all aspects of their customer relationships.

The Sugar application

It provides integrated management of corporate information on customer accounts and contacts, sales leads and opportunities, plus activities such as calls, meetings, and assigned tasks. The system seamlessly blends all of the functionality required to manage information on many aspects of your business into an intuitive and user-friendly graphical interface.

Sugar also provides a graphical dashboard to track the sales pipeline, the most successful lead sources, and the month-by-month outcomes for opportunities in the pipeline.

The Sugar community

Sugar is based on an open source project and therefore advances quickly through the development and contribution of new features by its supporting community.

Welcome to the community!



Audience

[The Sugar Developer Guide provides information for developers who want to extend and customize SugarCRM functionality using the customization tools and API's provided in the Sugar Community Edition, Sugar Professional Edition and Sugar Enterprise Edition.](#)

Feature Overview

[Sugar consists of modules which represent a specific functional aspect of CRM such as Accounts, Activities, Leads, and Opportunities. For example, the Accounts module enables you to create and manage customer accounts, while the Activities module enables you to create and manage activities related to accounts, opportunities, etc. Sugar modules are designed to help you manage your customer relationships through each step of their life cycle, starting with generating and qualifying leads, through the selling process, and on to customer support and resolving reported product or service issues. Because many of these steps are interrelated, each module displays related information. For example, when you view the details of a particular account, the system also displays the related contacts, activities, opportunities, and bugs. You can not only view and edit this information but can also create new information.](#)

[As a developer, Sugar gives you the ability to customize and extend functionality within the base CRM modules. You can customize the look and feel of Sugar across your organization. Or you can create altogether new modules and build entirely new application functionality to extend these new modules.](#)

Core Features

[Sales Force Automation](#)

- [Lead, Contact, and Opportunity Management to pursue new business, share sales information, track deal progress, and record deal-related interactions.](#)
- [Account management capabilities to provide a single view of customers across products, geographies, and status.](#)
- [Automated Quote and Contract management functionality to generate accurate quotes with support for multiple line items, currencies, and tax codes.](#)
- [Sales forecasting and pipeline analysis to give sales representatives and managers the ability to generate accurate forecasts based on sales data in Sugar.](#)



- [Sugar Dashboards to provide real-time information about leads, opportunities, and accounts.](#)
- [Sugar Plug-ins for Microsoft Office to integrate your CRM data with Microsoft's leading productivity tools.](#)
- [Sugar Mobile for iPhone, a native Sugar application for iPhone, to access contacts, opportunities, accounts, and appointments in Sugar Enterprise and Sugar Professional while logging calls and updating customer accounts.](#)
- [Sugar Mobile to access mobile functionality through any standards-based web browser.](#)

[Marketing Automation](#)

- [Lead management for tracking and cultivating new leads](#)
- [Email marketing for touching prospects and customers with relevant offers](#)
- [Campaign management for tracking campaigns across multiple channels](#)
- [Campaign Wizard to walk users through the process of gathering information such as the marketing channel, targets, and budget needed to execute a campaign effectively.](#)
- [Campaign reporting to analyze the effectiveness of marketing activities](#)
- [Web-to-Lead forms to directly import campaign responses into Sugar to capture leads.](#)

[Customer Support](#)

- [Case management to centralize the service history of your customers, and monitor how cases are handled.](#)
- [Bug tracking to identify, prioritize, and resolve customer issues](#)
- [Customer self-service portal to enable organizations to provide self-service capabilities to customers and prospects for key marketing, sales, and support activities.](#)



- [Knowledge Base to help organizations manage and share structured and unstructured information.](#)

Collaboration

- [Shared Email and calendar with integration to Microsoft Outlook](#)
- [Activity management for emails, tasks, calls, and meetings](#)
- [Content syndication to consolidate third-party information sources](#)
- [Sugar mobile functionality for wireless and PDA access for employees to work when they are away from the office.](#)
- [Sugar Offline Client to enable employees who work offline to update Sugar automatically when they return to the network.](#)

Reporting

- [Reporting across all Sugar modules](#)
- [Real-time updates based on existing reports](#)
- [Customizable dashboards to show only the most important information](#)

Administration

- [Edit user settings, views, and layouts quickly in a single location](#)
- [Define how information flows through Sugar \(workflow management\) and the actions users can take with information \(access control\)](#)
- [Customize the application in Studio to meets the exact needs of your organization.](#)
- [Create custom modules in Module Builder.](#)



Related Documentation

[Sugar Enterprise Application Guide](#), [Sugar Professional Application Guide](#), and [Sugar Community Edition Application Guide](#): Describe how to install, upgrade, set up, configure, manage, and use Sugar Enterprise, Professional, and Community Edition respectively.



Introduction

1. [Overview](#)
 2. [Background](#)
 3. [Application Framework](#)
 - 3.1. [Directory Structure](#)
 - 3.2. [Key Concepts](#)
 - 3.2.1. [Application Concepts](#)
 - 3.2.2. [Files](#)
 - 3.2.3. [Variables](#)
 - 3.2.4. [Entry Points](#)
 4. [Module Framework](#)
 5. [User Interface Framework](#)
 6. [Extension Framework](#)
 7. [Sugar Dashlets](#)
 8. [Web Services](#)
 9. [Cloud Connectors](#)
-

Overview

SugarCRM was originally written on the LAMP stack ([L](#)inux, [A](#)pache, [M](#)ySQL and [P](#)HP). Since version 1.0, the SugarCRM development team has added support for every operating system (including Windows, Unix and Mac OSX) on which the PHP programming language runs for the Microsoft IIS Web server, the Microsoft SQL Server, and Oracle databases. Designed as the most modern Web-based CRM platform available today, SugarCRM has quickly become the business application standard for companies around the world. See the [Supported Platforms](#) page for detailed information on supported software versions and recommended stacks.

Background

Sugar is available in three editions: the Community Edition, which is freely available for download under the GPLv3 public license, and the Professional and Enterprise Editions, which are sold under a commercial subscription agreement. All three editions are developed by the same development team using the same source tree with extra modules available in the Professional and Enterprise Editions. Sugar customers using the Professional and Enterprise Editions also have access to Sugar Support, Training, and Professional Services offerings. Contributions are happily accepted from the Sugar



Community, but not all contributions are included because SugarCRM maintains high standards for code quality.

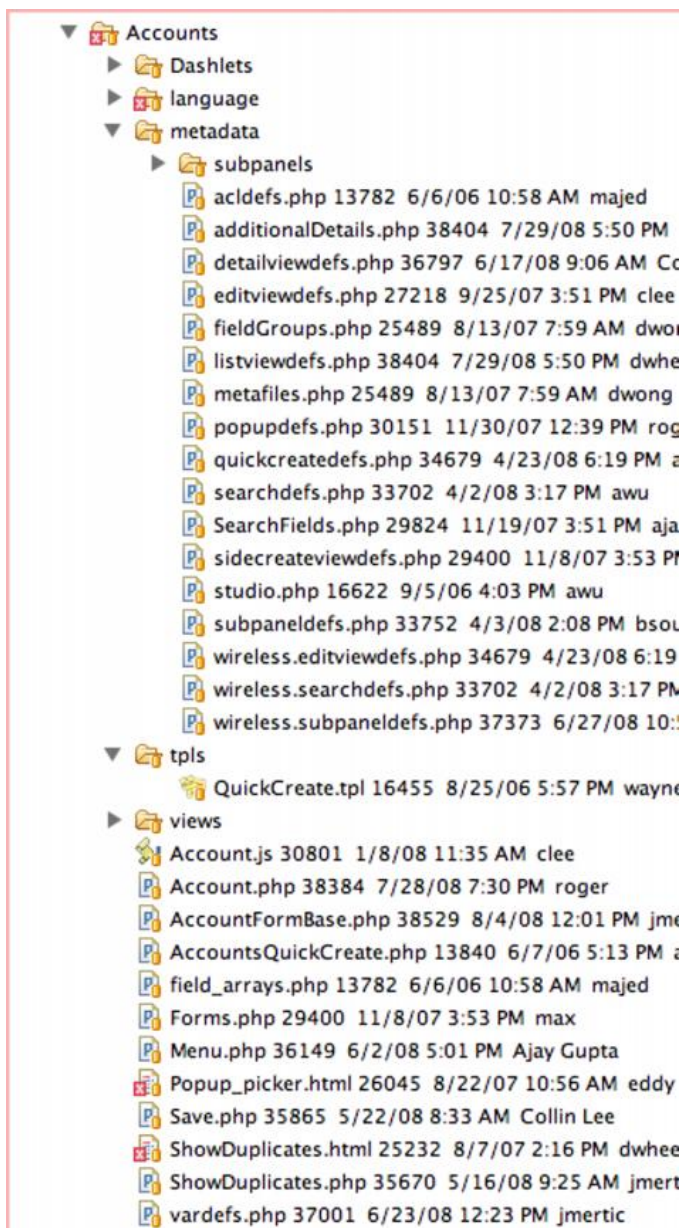
From the very beginning of the SugarCRM Open Source project in 2004, the SugarCRM development team designed the application source code to be examined and modified by developers. The Sugar application framework has a very sophisticated extension model built into it allowing developers to make significant customizations to the application in an upgrade-safe and modular manner. While it may be easy to modify one of the core files in the distribution, you should always check to see first if there is an upgrade-safe way to make your changes. Educating developers on how to make upgrade-safe customizations is one of the key goals of this Developer Guide.

Application Framework

The Sugar application code is based on a modular framework with secure entry points into the application (e.g. `index.php` or `soap.php`). All modules, whether core modules or ones you create and install through the Module Loader, must exist in the `<sugar root>/modules/` folder. Typically modules represent business entities or objects in Sugar such as 'Contacts', and the object has fields or attributes that are stored in the database, as well as a user interface (UI) for the user to create and modify records. A module encompasses definitions for the data schema, user interface, and application functionality.

The structure of Sugar's root directory is shown below.





Directory Structure

SugarCRM code resides in various directories within the Sugar installation. The structure of the directories within the Sugar application consists of the following root level directories:

- **cache:** Various cache files written to the file system to minimize database accesses and store user interface templates created from metadata. Also files uploaded into the application such as Note Attachments or Documents reside in this directory (refer to 'upload_dir' parameter in the config.php file) which means that this is an active cache directory, and not all files can be deleted from this directory.



- **custom:** Stores upgrade-safe customizations such as custom field definitions, user interface layouts, and business logic hooks.
- **data:** Key system files are located here, most notably the SugarBean base class which controls the default application logic for all business objects in Sugar.
- **include:** Many Sugar utility functions are located here, as well as other libraries that Sugar utilizes as part of its operations. Most notably in this directory is the utils.php file that contains the most widely used utility functions.
- **metadata:** This file contains relationship metadata for all many-to-many data relationships between the business objects.
- **modules:** This folder contains all modules in the system. Custom modules installed through the Module Loader exist here as well.

Key Concepts

These are the main files, classes and application concepts that comprise the Sugar platform.

Application Concepts

- **Controller:** Directs all incoming page requests. It can be overridden in each module to change the default behavior. It relies on Entry point parameters, described below, to serve the appropriate page.
- **Views:** A set of user interface actions managed by the Controller, the default views in Sugar include the Detail View, Edit View, and List View.
- **Display Strings:** Sugar is fully internationalized and localizable. Every language pack has its own set of display strings which is the basis of language localization. There are two types of display strings in the Sugar application: application strings and module strings. Application strings contain the user interface labels displayed globally throughout the application. The `$GLOBALS['app_strings']` array contains these labels. There is also the `$GLOBALS['app_list_strings']` array which contains the system-wide dropdown list values. Each language has its own application strings variables. The `$GLOBALS['mod_strings']` array contains strings specific to the current, or in-focus, module..
- **Dropdown Lists:** Dropdown lists are represented as `'name' => 'value'` array pairs located in the application strings as mentioned above. The `'name'` value is stored in the database where the `'value'` is displayed in the Sugar User Interface (UI). You are able to create and edit dropdown lists and their values through the UI, in Studio. For working with dropdown lists in Edit Views, use the handy `get_select_options_with_id()` utility function to help render the `<select>` input options. Also use the handy `translate()` utility



function for translating whatever string key you are working with into the user's currently selected display language.

Files

- **SugarBean.php:** This file located under the '<sugar root>/data' folder contains the SugarBean base class used by all business entity or module objects. Any module that reads, writes, or displays data will extend this class. The SugarBean performs all of the heavy lifting for data interactions, relationship handling, etc.
- **modules.php:** The modules.php file is a critical file in Sugar. It contains several variables that define which modules are active and usable in the application.

Variables

- **\$dictionary:** The \$dictionary array contains all module field variable definitions (vardefs), as well as the relationship metadata for all tables in the database. This array is dynamically built based upon the vardefs.php definitions.

Entry Points

The primary user interface entry point for Sugar is through **index.php** located in the root Sugar folder. There are three main parameters for most calls within Sugar, they are:

- **module:** The module to be accessed as part of the call
- **action:** The action to be taken by the application within the module
- **record:** The record ID.

An example URL for a Sugar call might be:

<http://www.yoursugarsite.com/index.php?module=Contacts&action=DetailView&record=d545d1dd-0cb2-d614-3430-45df72473cfb>

This URL invokes the Detail View action from within the Contacts module to display the record denoted by the record request value.

Other commonly used parameters are 'return_module', 'return_action', return_id'. This group of request parameters are typically used when a user selects to cancel out of an action such as when creating/editing a record.

Note: As of Sugar 5.1, most entry points were consolidated into index.php. Previous versions had other files as entry points into the application.



Module Framework

All modules, whether core modules or ones you create and install through the Module Loader are placed in the `<sugar root>/modules/` folder. Typically modules are created when you have to represent an object in Sugar, such as 'Contacts', and the object has data points that need to be stored in the database, as well as have an UI for the user to create, edit, and delete records for the object.

Let us look at a module's framework and how it fits into the overall Sugar Platform.

In the Platform Overview section, we show an example of a call that would be typical for a Detail View action within a particular module. There are five main actions for a module:

- **List View:** This Controller action exists to provide the user with the search form and search results for a module. From this screen, a user can take such actions as deleting, exporting, and mass updating multiple records or drill into a specific record to view and edit the details. Users can see this view by default when they click one of the module tabs at the top of the page. Files in each module describe the contents of the list and search view.
 - **Detail View:** A Detail View provides a read-only view of a particular object. Typically, a user will access a record's Detail View through the List View. The user can view the details of the object itself and will see, below the details, lists of related items that are referred to as 'sub-panels' in Sugar. Sugar panels act as mini List Views of items that are related to the parent object accessed with the Detail View action. For instance, Tasks assigned to a Project, or Contacts to an Opportunity will appear in sub-panels below the Project or Opportunity. `<module>/metadata/detailviewdefs.php` defines a module's Detail View page's layout. `<module>/metadata/subpaneldefs.php` defines the subpanels that are visible under the module's Detail View page.
 - **Edit View:** The Edit View action is accessed whenever a user is creating a new record or editing details of an existing one. It is possible to directly access the Edit View from the List View. `<module>/metadata/editviewdefs.php` defines a module's Edit View page layout.
 - **Save:** This Controller action is processed whenever the user clicks the 'Save' button from the record's Edit View.
 - **Delete:** This action is processed whenever the user clicks the 'Delete' button from the Detail View of a record or, as a special case, from an item listed in a sub-panel.

These actions are driven by the UI framework, and the framework relies on metadata files in the requested module.

- `<module>/metadata/listviewdefs.php` describes the layout of the List View.
- `<module>/metadata/searchdefs.php` describes the search form tabs above the List View.
- `<module>/metadata/editviewdefs.php` describes the layout of the Edit View.



- `<module>/metadata/detailviewdefs.php` describes the layout of the Detail View.

Besides the action files described above you probably have noticed some additional files located in the folder.

- **forms.php:** This file contains two functions to render specific JavaScript for validation or other actions you might want to perform during edits/saves. By default you can leave these empty and have them return ``;
- **Menu.php:** This file is responsible for rendering the Shortcuts menu, which was renamed as Actions menu as of Sugar 6.0. In Community Edition, the Actions menu displays below the module tabs and the Last Viewed bar. In Enterprise and Professional, the Actions menu displays on every module tab. By default, you usually add a link to create a new record, and a link to the List View to search.
- **Popup.php:** This file acts as a wrapper to the central Popup class located under the utils folder. It is called if ever another module wants to get a popup list of records from a related module. The central Popup class then uses the `Popup_picker.html` and `<MODULE_NAME>/metadata/popupdefs.php` file to render the popup.
- **Popup_picker.html:** Used by the central Popup class to display a module's popup.
- **vardefs.php:** The `vardefs.php` metadata file defines db and non-db fields for Sugar objects as well as relationships between objects.
- **field_arrays.php:** This file is deprecated as of Sugar version 5.1. It has been phased out over time with the addition of additional metadata structures in the application, most notably the `vardefs` metadata.

Now let us look at the various subfolders within a module folder to complete the overview of the module structure.





Sugar Dashlets: Sugar Dashlets are drag-and-drop forms displayed on the Sugar Home and Dashboard tabs. These forms can display any data, including data pulled from external connectors. With the Sugar default application, they contain List View and Chart data for the application modules. As a developer of a custom module you are able to create a Sugar Dashlet view to your new module. For each Sugar Dashlet you create, you will place the necessary files in the '<MODULE_NAME>/Dashlets' folder.

language: The language folder's main purpose is to hold the strings files for the module. By default you will have an *en_us.lang.php* file which contains ALL strings used specifically by your module. These strings are represented by the `$mod_strings` variable which can be accessed at any time after a global `$mod_string` call. The .html files located in this folder are used by the Help subsystem. Sugar provides the capabilities for multi-language support and the dynamic loading via the admin panel of new language packs.

metadata: As we have built more and more metadata and extensibility into the Sugar Platform, module-specific metadata files have been added to this folder. Some of the most important files in this directory include: *additionalDetails.php*, which defines the content of the popup displayed in the List Views; *listviewdefs.php*, which defines the columns displayed on the List View page; *popupdefs.php*, which defines the search fields and list columns for a module's popup; *SearchFields.php*, which defines the Basic Search and Advanced Search forms seen in the List View page; and *studio.php*, which defines how the Studio tool interacts with a module's metadata files.

subpanels: This folder will hold the definitions of a module's sub-panels when that module is related in a one-to-many or many-to-many fashion. Typically you have the *default.php* file. You can add any number of versions here. Alternatively, you can also create custom versions of sub-panels of other modules to be displayed by your custom module. For example, you can relate a custom module with the Contacts module and have a Contacts subpanel under the Detail View. For instance, you could build and place a '*ForWidgets.php*' file under the



'<sugar root>/modules/Contacts/subpanels/' folder. The file name is referenced by the 'subpanel_name' parameter called from a 'layout_defs.php' definition.

- **tpls:** This folder holds Smarty template files. Currently these are used for Quick Create forms.
- **views:** In this folder are files that can override the default Model-View-Controller (MVC) framework view files. View files can perform multiple different actions on the Smarty template or outputted HTML, allowing developers to modify and extend the default UI display classes and take full control of the user interface.

User Interface Framework

SugarCRM uses an implementation of the Model-View-Controller (MVC) pattern the base of all application interactions. Working closely with the MVC framework is a metadata-driven UI framework where the high-level specification of parts of the user interface in the application is described in a metadata structure.

Extension Framework

The extension framework in Sugar allows you to implement customizations of existing modules or create entirely new modules. Through the various extension framework capabilities you can extend most of the functionality of Sugar in an upgrade-safe manner. The Module Builder tool and Studio tool available in the Admin screen allows you to make the most common customizations outlined below. You can then further extend your system by adding upgrade-safe custom code. The areas open to extension are:

- **Modules:** You are to create entirely new modules and add them to Sugar.
- **Vardefs:** You are able to add custom fields to existing modules with the addition of your custom module.
- **Relationships:** New relationships can be added to the system between your new modules and existing modules in Sugar.
- **SubPanels:** With the addition of new relationships you can create/add new sub-panel definitions to existing modules.
- **Strings:** You can override or add to module and application strings.
- **Menus:** You can override or add to Shortcut menus.



- **Layout Defs:** You can specify the subpanels to display, and the order in which they are displayed in Sugar. Not only can you create the layout definition for a custom module, but you can also add it as a sub-panel to the layout definition of an existing module.

Sugar Dashlets

Sugar Dashlets is a framework that provides for Sugar Dashlet containers to be included in the Sugar UI. Sugar Dashlet container objects can display and interact with Sugar module data, with external sources such as RSS feeds, and with web services like Google Maps. Released originally in Sugar 4.5, Sugar Dashlets are a powerful new way to combine highly functional mash-ups in an attractive and easily tailored AJAX-based UI framework. Sugar Dashlets, located on the home page, allow for the customization through simple drag-and-drop tools. The Sugar Dashlet Framework allows developers to easily create new Sugar Dashlets that can be installed in SugarCRM instances through the Module Loader.

Web Services

Sugar provides a Web Services API interface for developers to build integrations with Sugar for reading and writing data. Sugar provides Web Services APIs through the NuSOAP PHP implementation of the SOAP and REST protocol. SOAP (Simple Object Access Protocol) is used for making Remote Procedure Calls through the HTTP protocol by relaying messages in XML. The SugarSoap API's, built on top of the NuSOAP PHP library, are included in the Sugar Community, Sugar Professional and Sugar Enterprise editions. REST (Representational State Transfer) is used for making method calls through HTTP protocol by sending and receiving messages in JSON/Serialize format. Framework supports the addition of multiple formats for REST. For example, you can add XML format to send and receive data.

Cloud Connectors

The Cloud Connector framework enables developers to integrate data from external Web Services and widgets into their Sugar installation. Data from existing modules such as accounts, contacts, and leads may act as inputs to retrieve external data.

For Community Edition, Sugar supports LinkedIn®'s Company Insider widget. You can use this as an example connector to learn the framework and create your own. Sugar Professional and Sugar Enterprise also support additional connectors, such as Hoovers® and Zoominfo, and have the ability to merge the data into existing Sugar records.

The main components for the framework are the factories, source, and formatter classes.

The factories are responsible for returning the appropriate source or formatter instance for a connector. Sources are responsible for encapsulating the retrieval of data as a single record, or a list, or records of



the connectors. Formatters are responsible for rendering the display elements of the connectors. For more information, see [Chapter 4 Customizing Sugar.docx](#).

Copyright 2004-2010 SugarCRM Inc.
[Community Edition License](#)



Application Framework

1. [Overview](#)
2. [Entry points](#)
3. [Upgrade implications](#)
 - 3.1. [Backwards Compatibility with Custom Code](#)
4. [File Caching](#)
5. [Sugar Dashlets](#)
 - 5.1. [Sugar Dashlet Files](#)
 - 5.2. [Templating](#)
 - 5.3. [Categories](#)
 - 5.4. [Sugar Dashlet Base Class](#)
 - 5.5. [Sugar Dashlets JavaScript](#)
6. [Browser JavaScript](#)
 - 6.1. [Accessing Language Pack Strings](#)
 - 6.2. [Quicksearch](#)
 - 6.2.1. [Requirement for a QuickSearch Field:](#)
7. [ACL](#)
8. [Scheduler](#)
9. [Databases](#)
 - 9.1. [Indexes](#)
 - 9.2. [Primary Keys, Foreign Keys, and GUIDs](#)
10. [Logger](#)
 - 10.1. [Logger Level](#)
 - 10.2. [Log File Name](#)
 - 10.3. [Log File Extension](#)
 - 10.4. [Log File Date Format](#)
 - 10.5. [Max Log File Size](#)
 - 10.6. [Max Number of Log Files](#)
 - 10.7. [Log Rotation](#)
 - 10.8. [Custom Loggers](#)
11. [Web Services](#)
 - 11.1. [SOAP](#)
 - 11.1.1. [SOAP Protocol](#)
 - 11.2. [REST](#)
 - 11.2.1. [REST Protocol](#)



- 12. [API Definitions](#)
 - 12.1. [Core Calls](#)
 - 12.2. [Extensibility in Upgrade Safe Manner](#)
 - 12.3. [SOAP Errors](#)
 - 12.4. [SugarSoap Examples](#)
- 13. [Cloud Connectors Framework](#)
 - 13.1. [Factories](#)
 - 13.2. [Sources](#)
 - 13.3. [Formatters](#)

Overview

The Sugar application code is based on a modular framework with secure entry points into the application (e.g. index.php or soap.php). All modules, core or custom, must exist in the <sugar root="">/modules/ folder. Modules represent business entities or objects in Sugar such as Contacts, and the object has fields or attributes that are stored in the database, as well as a user interface (UI) for the user to create and modify records. A module encompasses definitions for the data schema, user interface, and application functionality.

Entry points

All entry points into the Sugar application are pre-defined to ensure that proper security and authentication steps are applied consistently across the entire application.

- `campaign_tracker.php` – used by the Campaign Management module for tracking campaign responses. Deprecated as of Sugar 5.1.0.
- `cron.php` – used by the Windows Scheduler Service or the cron service on Linux and Unix for executing the Sugar Scheduler periodically.
- `index.php` – default entry point into the Sugar application
- `install.php` – used for initial install
- `maintenance.php` – invoked when the application is down for maintenance.
- `metagen.php` - Deprecated as of Sugar 5.1.0.
- `silentUpgrade.php` – used for silent installer



- soap.php – entry point for all SOAP calls
- vcal_server.php – used for syncing information to Outlook

Upgrade implications

One of the many code re-factoring changes we made as of Sugar 5.1, was to consolidate the number of entry points into the application as well as re-routing the current entry points through the MVC framework. An entry point is a PHP file that can be called either through the URL or the command line to invoke a Sugar process. For instance, calling the home page of the application through the URL, or starting the Scheduler through the command line. Consolidating the entry points has also helped us secure the application better and improve quality by making sure each request goes through the same initialization code.

Backwards Compatibility with Custom Code

It does, however, present some backwards compatibility problems. Most notably, you will need to update your code if you have custom code that relies on a deprecated entry point such as a custom Quote template that may have called pdf.php, which is no longer a stand-alone entry point. In such cases, you will need to change the URL reference as described below:

1. Look for the entry point file name in the include/MVC/Controller/entry_point_registry.php. It will be in the 'file' key in the sub-array. Make note of the key of that array
2. For the pdf.php entry point, the array appears in include/MVC/Controller/entry_point_registry.php as:

```
'pdf' => array('file' => 'pdf.php', 'auth' => true),
```

We will need to use the 'pdf' part in the next step.

3. Change the URL reference from the current reference to one in the form of:

```
index.php?entryPoint=<<entrypoint>>
```

For the above pdf.php example, we would change our references from:

```
http://<your site>/pdf.php
```

to

```
http://<your site>/index.php?entryPoint=pdf
```



The only remaining entry point that is not using this new index.php URL pattern (and, therefore, continues to be a valid entry point) is:

- `campaign_tracker.php` – used by the Campaign Management module for tracking campaign responses. Deprecated as of Sugar 5.1.0.
- `cron.php` – used by the Windows Scheduler Service or the cron service on Linux and Unix for executing the Sugar Scheduler periodically.
- `index.php` – default entry point into the Sugar application
- `install.php` – used for initial install
- `maintenance.php` – invoked when the application is down for maintenance.
- `metagen.php` – Deprecated as of Sugar 5.1.0.
- `silentUpgrade.php` – used for silent installer
- `soap.php` – entry point for all SOAP calls
- `vcal_server.php` – used for syncing information to Outlook

File Caching

Much of the user interface is built dynamically using templates from metadata and language string files. SugarCRM implements a file caching mechanism to improve the performance of the system by reducing the number of static metadata and language files that need to be resolved at runtime. This directory stores the cached template and language string files.

When developing in Sugar, it is recommended that you turn on the **Developer Mode** (Admin->System Settings->Advanced->Developer Mode), which causes the system to ignore these cached files. This is especially helpful when you are directly altering templates, metadata, or language files. When using Module Builder or Studio, the system will automatically refresh the file cache. Turn off the Developer Mode when you have completed your customizations because this mode degrades system performance.



Sugar Dashlets

Sugar Dashlets use the [abstract factory](#) design pattern. Individual Dashlets extend the base abstract class Dashlet.php, List View Dashlets extend the base abstract class DashletGeneric.php, while chart Dashlets extend the base abstract class DashletGenericChart.php.

Sugar Dashlet instances must be contained in one of the following directories:

- modules/moduleName/Dashlets/
- custom/modules/moduleName/Dashlets/

Typically, Sugar Dashlet developers will want to use the *custom/* directory in order to make their Sugar Dashlets upgrade-safe. The standard *modules/* directory location is where you'll find Sugar Dashlets offered in base Sugar releases.

Sugar Dashlet Files

The file name containing the main Sugar Dashlet code must match the Sugar Dashlet's class name. For example, the Sugar Dashlet class *JotPadDashlet* will be found in the file */Home/Dashlets/JotPadDashlet/JotPadDashlet.php*. The *JotPadDashlet* Dashlet is a sample Sugar Dashlet released originally in Sugar 4.5. It serves as a useful example from which to begin your development efforts.

A metadata file accompanies each Sugar Dashlet. It contains descriptive information about the Sugar Dashlet as defined here:

```
$DashletMeta['JotPadDashlet'] = array
(
'title' => 'LBL_TITLE',
'description' => 'LBL_TITLE',
'icon' => 'themes/Sugar/images/Accounts.gif',
'category' => 'Tools'
);
```

The naming convention for the metadata file is *className.meta.php*, where *className* is the name of your Sugar Dashlet. It must appear in the same directory as the Sugar Dashlet code. For *JotPad Dashlet*, the meta file is stored in

modules/Home/Dashlets/JotPadDashlet/JotPadDashlet.meta.php.

The 'title' and 'description' elements are translated. If the values here match a key in the array \$DashletStrings (from the language file) then they will be translated, otherwise it will display the literal string. (It is a best practice to use translatable language strings so that your Sugar Dashlet is international!)



Language files have a similar naming convention: *className.locale.lang.php* (e.g., */Dashletsmodules/Home/Dashlets/JotPadDashlet/JotpadDashlet.en_us.lang.php*)

Icon files can either be defined in the *.metadata* file or simply included in the Sugar Dashlet Directory (e.g., */Dashletsmodules/Home/Dashlets/JotPadDashlet/JotPadDashlet.icon.png*). The system will scan for image files in the corresponding Sugar Dashlet directory.

Templating

The suggested templating engine for Sugar Dashlets is [Smarty](#), however it is not a requirement.

Categories

There are five categories for Sugar Dashlets.

- **Module Views** – Generic views of data in modules
- **Portal** – Sugar Dashlets that allow access to outside data (RSS, Web services, etc)
- **Charts** – Data charts
- **Tools** – Various tools such as notepad, calculator, or even a world clock!
- **Miscellaneous** - Any other Sugar Dashlet

Sugar Dashlet Base Class

The main Sugar Dashlet base class is *include/Dashlets/Dashlet.php*. All Sugar Dashlets should extend.

You must assign each Sugar Dashlet a unique ID. This ID is used in the HTML document when it is displayed. Unique IDs allow multiple Sugar Dashlets of the same type to be included on the page.

Sugar Dashlets are stored in the table *user_preferences* under the name 'Dashlets' and the category 'home'.

The 'options' element stores the options for the Sugar Dashlet. This element is loaded/stored by *storeOptions* / *loadOptions* functions in the base Dashlet class.

Sugar Dashlets JavaScript

Sugar Dashlet utility functions are located in *include/JavaScript/Dashlets.js*. This contains the following methods:

```
postForm: function(theForm, callback) {}
```



postForm method is used to post the configuration form through AJAX. The callback will usually be SUGAR.sugarHome.uncoverPage to remove the configuration dialog.

```
callMethod: function(DashletId, methodName, postData, refreshAfter, callback) {}
```

callMethod is a generic way to call a method in a Dashlet class. Use this function to generically call a method within your Dashlet class (php side). Optionally, you can refresh your Dashlet after a call and also utilize a callback function.

This method can also be used as a way to proxy AJAX calls to Web services that do not exist in the SugarCRM installation, for example, Google Maps Mash-up.

Browser JavaScript

Because Sugar is a Web-based application, executing custom logic on the client-side (for example, validating data before posting it back to the server) requires writing JavaScript. This section outlines the JavaScript constructs available to the developer.

In order to improve performance, Sugar's production JavaScript files are compressed with the JSMIn library. This process reduces JavaScript file sizes, thereby reducing download times. The originally formatted JavaScript files are in the /jssource directory. It is a best practice to make any JavaScript code changes in the /jssource/src_files folders and then use the "Rebuild JS Compressed Files" option in Admin->Repair.

Accessing Language Pack Strings

All language pack strings are accessible within the browser-side JavaScript. To access these strings simply use the following JavaScript call:

```
// LBL_LOADING string stored in $app_strings
```

```
SUGAR.language.get('app_strings', 'LBL_LOADING');
```

```
// LBL_LIST_LAST_NAME string stored in Contacts $mod_strings
```

```
SUGAR.language.get('Contacts', 'LBL_LIST_LAST_NAME');
```

Admins and translators will need to be aware that these JavaScript language files are cached. If there are any changes to the language files they will need to 'rebuild' the JavaScript files from the Repair console in the Admin section. This essentially removes the cache files and they will be rebuilt when needed. It also increments `js_lang_version` in `sugar_config` so that user's browser will re-cache these js files.



Quicksearch

As of version 5.1, Sugar uses a type-ahead combo box system called "QuickSearch" that is based around [ExtJS](#). The Sugar QuickSearch (SQS) code resides in the file `<sugar_root>/include/javascript/quicksearch.js`. The ExtJS library which drives the QuickSearch is located at `<sugar_root>/include/javascript/ext-2.0/ext-quicksearch.js`. These two files are grouped in `<sugar_root>/include/javascript/sugar_grp1.js` which is loaded in all the main page of SugarCRM. A custom Ext ComboBox is used to pull data through an AJAX call to <http://yourserver/SugarCRM/index.php> which then accesses the file `<sugar_root>/modules/Home/quicksearchQuery.php`.

The browser will initiate an AJAX call through JavaScript to the server 700ms after the last user input takes place in the browser. A call is then made requesting up to 30 results per result set.

The first twelve results are then displayed in the browser. If the user refines the search, and the result set is a subset of the first call then no additional call is made. If the result set of the first call is equal to the limit (30), then an additional call will be made.

Requirement for a QuickSearch Field:

- ? The class of the field must be set to "sqsEnabled".
- ? The field must not be set to "disabled" or "readOnly".
- ? The "sqs_objects" JS array must be defined and must contain the field name.
- ? "sugar_grp1.js" must be loaded on the page.

Custom Parameter :

sqsNoAutofill : add this string to the class of the field to disable the Automatic filling of the field on Blur.

Metadata example:

```
array(  
    'name' => 'account_name',  
    'displayParams' => array(  
        'hideButtons'=>'true',  
        'size'=>30,  
        'class'=>'sqsEnabled sqsNoAutofill'  
    )  
)
```

ACL

ACLs, or Access Control Lists, are used to restrict access to Sugar modules, and the data and actions available to users within Sugar modules (for example, "Delete" and "Save"). ACLs are defined in the



Roles area of Sugar Admin. Sugar Professional and Enterprise Editions restrict user access down to specific fields.

You can check whether the current user has access to a particular action using the following code:

```
if (ACLController::checkAccess($category, $action, $is_owner, $type)) {  
  
    // your code here  
  
}
```

Where the parameters mean the following:

- `$category` = this corresponds to the module directory where the bean resides. For example: Accounts
- `$action` – the action you want to check against. For example: Edit. These actions correspond to actions in the `acl_actions` table as well as actions performed by the user within the application.
- `$is_owner` – whether or not the owner of the record attempting an action. Defaults to false. This only comes into play when the access level = `ALLOW_OWNER`
- `$type` = this defaults to `'module'`.

See the “Roles” section in the *Sugar Application Guide* for a list of actions and their possible values.

Scheduler

SugarCRM contains a Scheduler service that executes predefined functions asynchronously on a periodic basis. The Scheduler integrates with external UNIX systems and Windows systems to run jobs that are scheduled through those systems. The typical configuration is to have a UNIX cron job or a Windows scheduled job execute the SugarCRM Scheduler service every couple minutes. The Scheduler service checks the list of Schedulers defined in the Scheduler Admin screen, and executes any that are currently due.

A series of Schedulers are defined by default with every SugarCRM installation such as “Process Workflow Tasks” and “Run Report Generation Scheduled Tasks”.



Scheduler	Interval	Range	Status
Process Workflow Tasks	As often as possible	2005-01-01 16:30 - 2020-12-31 22:59	Active
Run Report Generation Scheduled Tasks	On the hour, 06:00	2005-01-01 11:30 - 2020-12-31 22:59	Inactive
Prime Tracker Tables	On the hour, 02:00, 1st	2005-01-01 08:30 - 2020-12-31 22:59	Active
Check Inbound Mailboxes	As often as possible	2005-01-01 05:15 - 2020-12-31 22:59	Active
Run Nightly Process Bounced Campaign Emails	On the hour, From 02:00 to 06:00	2005-01-01 09:00 - 2020-12-31 22:59	Active
Run Nightly Mass Email Campaigns	On the hour, From 02:00 to 06:00	2005-01-01 18:45 - 2020-12-31 22:59	Active
Prime Database on 1st of Month	On the hour, 04:00, 1st	2005-01-01 08:15 - 2020-12-31 22:59	Inactive
Update tracker_sessions table	As often as possible	2005-01-01 18:00 - 2020-12-31 22:59	Active

Databases

Sugar Enterprise, Sugar Professional, and Sugar Community Edition support the MySQL and Microsoft SQL Server databases. Additionally, Sugar Enterprise also supports the Oracle database. In general, Sugar uses only common database functionality, and the application logic is embedded in the PHP code. This means that Sugar uses no database triggers or stored procedures. This design simplifies coding and testing across different database vendors. Therefore, typically, the only implementation difference that you will find across the various supported databases is column types.

Sugar uses the mysql PHP extension for MySQL support (or mysqli if it enabled), the mssql extension for Microsoft SQL Server support, and the oci8 extension for Oracle support. Sugar does not support generic ODBC access or other database drivers such as PDO.

Indexes

Indexes can be defined directly in the main or custom vardefs.php for module, in an array under the key 'indices'. Below is the example of defining several indices:

```
'indices' => array (
    array(
        'name' => 'idx_modulename_name',
        'type' => 'index',
        'fields' => array('name'),
    ),
    array(
        'name' => 'idx_modulename_assigned_deleted',
        'type' => 'index',
        'fields' => array('assigned_user_id', 'deleted'),
    ),
),
```

The name of the index must start with idx_, and must be unique across the database. The possible values for 'type' include 'primary' for a primary key or 'index' for a normal index. The fields list matches the column names used in the database.



Primary Keys, Foreign Keys, and GUIDs

By default, Sugar uses globally unique identification values (GUIDs) for primary keys for all database records. Sugar provides a `create_guid()` utility function for creating these GUIDs in the following format: aaaaaaa-bbbb-cccc-dddd-eeeeeeeeeeee. The primary key column length is 36 characters.

The GUID format and value has no special meaning in Sugar other than the ability to match records in the DB. Sugar links two records (such as an Account record with a Contact record) with a specified ID in the record type relationship table (e.g. `accounts_contacts`).

Sugar allows a primary key to contain any unique string. This can be a different GUID algorithm, a key that has some meaning (such as bean type first, followed by info), an external key, and/or auto-incrementing numbers (converted to strings). Sugar chose GUIDs rather than auto-incrementing keys to allow for easier data synchronization across databases (in order to avoid primary key collisions). This data synchronization issue comes into play when the Sugar Offline Client (part of Sugar Enterprise) syncs data between the main Sugar installation and the Offline Client, or when developers use the Sugar SOAP APIs or a tool like Talend for data synchronization.

The reason the Offline Client uses GUIDs for primary keys is because it is very easy to implement, and handling data conflicts is simpler than with other schemes. If a developer changes Sugar to use some other ID scheme and does need to accommodate data synchronization across data stores, then he would have to either partition the IDs ahead of time or work out a system similar to the Sugar implementation for Cases, Quotes, and Bugs. For these modules, which have human-readable ID numbers (integers) that need to be synchronized across databases, Sugar implements a server ID that is globally unique and concatenates it with an incrementing Case, Quotes or Bug number. Attempting such a change to Sugar, while possible, would require some careful planning and implementation.

However if the developer does not need to worry about data synchronization issues, then he can certainly change the primary key format to some other unique string.

You can also import data from a previous system with one primary key format and then have all new records in Sugar use the GUID primary key format. All keys simply need to be stored as unique strings with no more than 36 characters.

To implement a new primary key method, or to import existing data with a different primary key format and then rely on the existing GUID mechanism for new records, there are a few things to look out for:

- The system expects primary keys to be string types and will format the SQL with quotes. If you change the primary key types to an integer type, you might have SQL errors to deal with since Sugar stores all ID values in quotes in our generated SQL. The database may be able to ignore this issue. MySQL running in Safe mode will have issues, for instance.
- Case-sensitivity can be an issue. IDs "abc" and "ABC" are typically treated the same in MySQL but represent different values in Oracle. When migrating data to Sugar, some CRM systems may use case sensitive strings as their IDs on export. If this is the case, and you are running MySQL, you need to run an algorithm on the data to make sure all of the IDs are unique. One simple



algorithm is to MD5 the ids that they provide. A quick check will let you know if there is a problem. If you imported 80,000 leads and there are only 60,000 in the system, some may have been lost due to non-unique primary keys, as a result of case sensitivity.

Sugar only tracks the first 36 characters in the primary key. Any replacement primary key will either require changing all of the ID columns with one of an appropriate size or to make sure you do not run into any truncation or padding issues. MySQL in some versions has had issues with Sugar where the IDs were not matching because it was adding spaces to pad the row out to the full size. MySQL's handling of char and varchar padding has changed in some of the more recent versions. To protect against this, you will want to make sure the GUIDs are not padded with blanks in the DB.

Logger

The Sugar Logger allows developers and system administrators to log system events and debugging information into a log file. The Sugar code contains log statements at various points, which are triggered based on the logging level.

For example, to write a message to the `sugarcrm.log` file when the log level is set to `'fatal'`, add the following in your code:

```
$GLOBALS['log']->fatal('my fatal message');
```

Logger Level

The logger level determines how much information is written to the `sugarcrm.log` file. You will find the `sugarcrm.log` file in the root of your Sugar installation.

Valid values are `'debug'`, `'info'`, `'error'`, `'fatal'`, `'security'`, and `'off'`. The logger will log information for the specified *and* higher logging levels. For example if you set the log level to `'error'` then you would see logs for `'error'`, `'fatal'`, and `'security'`. You also may define your own log levels on the fly. For example if you set the value to `'test'` then only values logged to `$GLOBALS['log']->test('This is my test log message');` would be logged. You should avoid using the logging level of `'off'`. The default value is `'fatal'`.

```
$GLOBALS['sugar_config']['logger']['level'] = 'debug';
```

You can also force the log level in your code by using:

```
$GLOBALS['log']->setLevel('debug');
```

Log File Name

You may concatenate static strings, variables, and function calls to set this value. For example if you wish to have monthly logs set this to `'sugarcrm' . date('Y_m')` and every day it will generate a new log file. The default value is `'sugarcrm'`.



```
$GLOBALS['sugar_config']['logger']['file']['name']
```

Log File Extension

The default value is '.log'. Therefore the full default log file name is 'sugarcrm.log'.

```
$GLOBALS['sugar_config']['logger']['file']['ext']
```

Log File Date Format

The date format for the log file is any value that is acceptable to the PHP `strftime()` function. The default is '%c'. For a complete list of available date formats, please see the [strftime\(\)](#) PHP documentation.

```
$GLOBALS['sugar_config']['logger']['file']['dateformat']
```

Max Log File Size

This value controls the max file size of a log before the system will roll the log file. It must be set in the format '10MB' where 10 is number of MB to store. Always use MB as no other value is currently accepted. To disable log rolling set the value to false. The default value is '10MB'.

```
$GLOBALS['sugar_config']['logger']['file']['maxSize']
```

Max Number of Log Files

When the log file grows to the 'maxSize' value, the system will automatically roll the log file. The 'maxLogs' value controls the max number of logs that will be saved before it deletes the oldest. The default value is 10.

```
$GLOBALS['sugar_config']['logger']['file']['maxLogs']
```

Log Rotation

The Sugar Logger will automatically rotate the logs when the 'maxSize' has been met or exceeded. It will move the current log file to <Log File Name> . 1 . <Log Extension>. If <Log File Name> . 1 . <Log Extension> already exists it will rename it to <Log File Name> . 2 . <Log Extension> prior. This will occur all the way up to the value of 'maxLogs'.

Custom Loggers

You can also extend the logger to integrate a different logging system into SugarCRM. For example, you can write log entries to a centralized application management tool, or write messages to a developer tool such as FirePHP.

To do this, you simply can create a new instance class that implements the `LoggerTemplate` interface. The below code is an example of how to create a FirePHP logger.

```
class FirePHPLogger implements LoggerTemplate
{
/**
```



```

* Constructor
*/
public function __construct()
{
if ( isset($GLOBALS['sugar_config']['logger']['default'])
&& $GLOBALS['sugar_config']['logger']['default'] == 'FirePHP' )
LoggerManager::setLogger('default','FirePHPLogger');
}
/**
* see LoggerTemplate::log()
*/
public function log(
$level,
$message
)
{
// change to a string if there is just one entry
if ( is_array($message) && count($message) == 1 )
$message = array_shift($message);
switch ( $level )
{
case 'debug':
FB::log($message);
break;
case 'info':
FB::info($message);
break;
case 'deprecated':
case 'warn':
FB::warn($message);
break;
case 'error':
case 'fatal':
case 'security':
FB::error($message);
break;
}
}
}
}

```

You can specify which log levels this backend can use in the constructor by calling the `LoggerManager::setLogger()` method and specifying the level to use for this logger in the first parameter; passing `'default'` makes it the logger for all logging levels. The only method needing implementing is the `log()` method, which writes the log message to the backend. To have this logger used, put it in the `/custom/include/SugarLogger/` directory with the naming `classname.php`.

Web Services



SOAP

SugarCRM provides Web Services API's through the NuSOAP PHP implementation of the SOAP protocol. SOAP stands for "Simple Object Access Protocol." It is used for making Remote Procedure Calls through the HTTP protocol by relaying messages in XML.

The SugarSoap API's are built using the NuSOAP library and included in the Sugar Community Edition, Sugar Professional Edition and Sugar Enterprise Edition. You can view the SugarSoap API's at:

<http://www.example.com/sugar/service/v2/soap.php>

The SugarSoap WSDL (Web Services Description Language) is located at:

<http://www.example.com/sugar/service/v2/soap.php?wsdl>

Thanks to a myriad of toolkits, it is easy to make effective use of SOAP Web Services from a variety of programming languages, in particular with Sugar and the wide array of SOAP-based services that it offers.

For these exercises, we will use PHP in conjunction with the NuSOAP Toolkit. You could, of course, connect to SugarCRM through SOAP and write your application code in Java, C++ or a variety of other SOAP-enabled programming languages.

Note: If the Sugar config.php variables site_url is wrong, SugarSoap will not work. Be sure to verify this value before continuing.

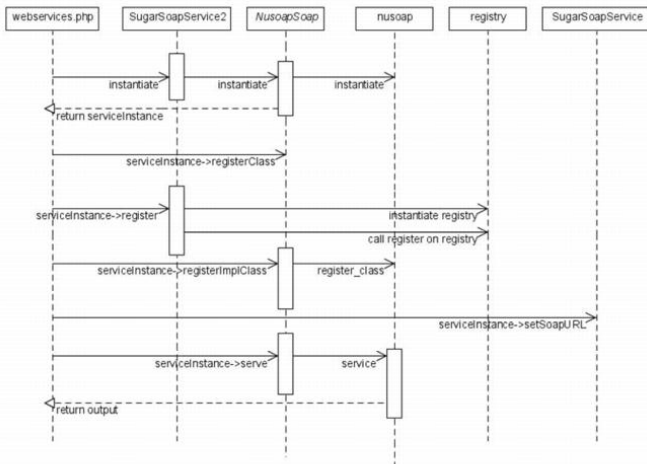
SOAP Protocol

SOAP, a standard Web Services protocol implementation, is a way of exchanging structured information of application functionality. The URL to SOAP is <http://localhost/service/v2/soap.php>, and the URL for WSDL is <http://localhost/service/v2/soap.php?wsdl>. The WSDL file contains the descriptions for all methods with input/output datatype.

See *examples/SoapTestPortal2.php* for more examples on usage.

Following is the complete SOAP flow.





REST

SugarCRM provides APIs through REST implementation. You can view SugarREST APIs at:

<http://www.example.com/sugar/service/v2/rest.php>

You can use `/service/utils/SugarRest.js` to make rest calls using javascript. Look at `/service/test.html` for examples on how to make REST calls from javascript. You can also use curl module to make REST call from server side. Look at the following example

```

$url = $sugar_config['site_url'] . "/service/v2/rest.php";
$result = doRESTCALL($url,
'login',array('user_auth'=>array('user_name'=>$user_name,'password'=>md5($user_password),
'version'=>'.01'), 'application_name'=>'SoapTest', 'name_value_list' => array(array('name' =>
'notifyonsave', 'value' => 'false'))));
function doRESTCALL($url, $method, $data) {
ob_start();
$ch = curl_init();
$headers = (function_exists('getallheaders'))?getallheaders(): array();
$_headers = array();
foreach($headers as $k=>$v){
$_headers[strtolower($k)] = $v;
}
// set URL and other appropriate options
curl_setopt($ch, CURLOPT_URL, $url);
curl_setopt ($ch, CURLOPT_POST, 1);
curl_setopt($ch, CURLOPT_HTTPHEADER, $_headers);
curl_setopt($ch, CURLOPT_HEADER, 1);
curl_setopt($ch, CURLOPT_SSL_VERIFYPEER, 0);
curl_setopt($ch, CURLOPT_RETURNTRANSFER, 1);
curl_setopt($ch, CURLOPT_FOLLOWLOCATION, 0);
curl_setopt($ch, CURLOPT_HTTP_VERSION, CURL_HTTP_VERSION_1_0 );
$post_data = 'method=' . $method . '&input_type=json&response_type=json';
$json = getJSONObj();
$jsonEncodedData = $json->encode($data, false);
  
```



```
$post_data = $post_data . "&rest_data=" . $jsonEncodedData;
curl_setopt($ch, CURLOPT_POSTFIELDS, $post_data);
$result = curl_exec($ch);
curl_close($ch);
$result = explode("\r\n\r\n", $result, 2);
print_r($result[1]);
$response_data = $json->decode($result[1]);
ob_end_flush();
//print_r($response_data);
return $response_data;
```

REST Protocol

Sugar uses the REST protocol to exchange application information. The URL for REST is <http://localhost/service/v2/rest.php>. The input/output datatype for REST is JSON/PHP serialize. These datatype files, *SugarRestJSON.php* and *SugarRestSerialize.php*, are in *service/core/REST* directory.

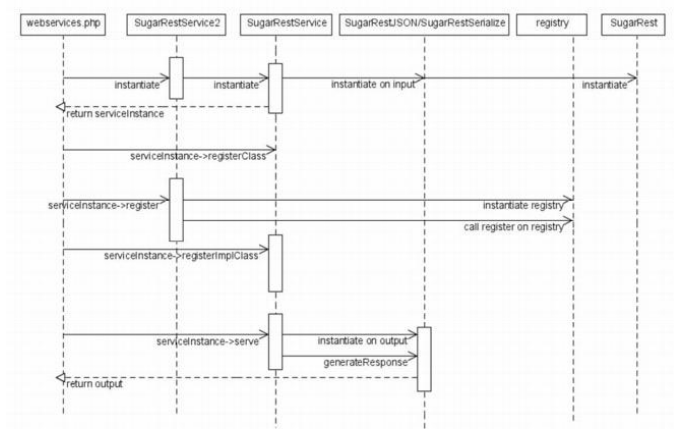
You can also define you own datatype. To do this, you need to create a corresponding *SugarRestCustomDataType.php* file in the *service/core/REST* directory and override `generateResponse()` and `serve()` functions.

The `Serve` function decodes or unserializes the appropriate datatype based on the input type; the `generateResponse` function encodes or serializes it based on the output type. See *service/test.html* for more examples on usage. In this file, the `getRequestData` function, which generates URL with json, is both the `input_type` and the `response_type`. That is, the request data from the javascript to the server is json and response data from server is also json. You can mix and match any datatype as input and output. For example, you can have json as the `input_type` and `serialize` as the `response_type` based on your application's requirements.

Sugar also provides an example on how to use REST protocol to retrieve data from other Sugar instances. In the example, *service/example.html* uses the *SugarRest.server_url* variable to make a request to *Rest_Proxy.ph*, which redirects this request to the appropriate Sugar instance and sends the response back to *service/example.html*. This *server_url* variable is a REST URL to other Sugar instances from which you want to retrieve data.

The REST flow is shown below.





API Definitions

SugarCRM provides an application programming interface, Sugar Web Services API, to enable you to customize and extend your Sugar application. As of version 5.5.0, Sugar provides an API for enhanced performance and provides versioning to help extend your Sugar application in an upgrade-safe manner.

Versioning

All API classes are located in the Service directory. The URL for the Service directory is <http://localhost/service/v2/soap.php>. The Service directory contains the following folders:

- core – A directory containing the core classes.
- REST - A directory containing REST protocols (example, JOSN and PHP serialize classes).
- V2 – A directory containing version-specific classes for SOAP and REST implementation. This folder contains the following variables:

- \$webservice_class – service class responsible for soap services - SugarSoapService2
- \$webservice_path – The location of the service class - V2/SugarSoapService2.php
- \$registry_class – Responsible for registering all the complex data types and functions available to call - registry
- \$registry_path – The location of registry class - service/v2/registry.php
- \$webservice_impl_class – The implementation class for all the functions - SugarWebServiceImpl
- \$location – The location of soap.php - '/service/v2/soap.php';
- \$soap_url – The URL to invoke - \$GLOBALS['sugar_config']['site_url'].'/service/v2/soap.php';

Call webservices.php – core/webservice.php is the file which is responsible for instantiating service class and calling different helper methods based on the values of above variables



Core Calls

The supported calls in the API are listed and described in this section.

Call	Description
Call: get_entry()	Retrieves a single SugarBean based on the ID.
Call: get_entries()	Retrieves multiple SugarBeans based on IDs. This API is not applicable to the Reports module.
Call: get_entry_list()	Retrieves a list of SugarBeans.
Call: set_relationship()	Sets a single relationship between two beans where items are related by module name and ID.
Call: set_relationships()	Sets multiple relationships between two beans where items are related by module name and ID.
Call: get_relationship()	Retrieves a collection of beans that are related to the specified bean and, optionally, return relationship data for the related beans.
Call: set_entry()	Creates or updates a single SugarBean.
Call: set_entries()	Creates or updates a list of SugarBeans.
Call: login()	Logs the user into the Sugar application.
Call: logout()	Logs out the user from the current session.
Call: get_server_info()	Obtains server information such as version and GMT time.
Call: get_user_id()	Returns the user_id of the user who is logged into the current session.
Call: get_module_fields()	Retrieves the vardef information on fields of the specified bean.
Call: seamless_login()	Performs a seamless login. This is used internally during synchronization.
Call: set_note_attachment()	Adds or replaces an attachment to a note.
get_note_attachment()	Retrieves an attachment from a note.



Call: set_document_revision()	Sets a new revision to the document
Call: get_document_revision()	Allows an authenticated user with the appropriate permission to download a document.
Call: search_by_module()	Returns the ID, module_name, and fields for the specified modules as specified in the search string.
Call: get_available_modules()	Retrieves the list of modules available to the current user logged into the system.
Call: get_user_team_id()	Retrieves the ID of the default team of the user who is logged into the current session.
Call: set_campaign_merge()	Performs a mail merge for the specified campaign.
Call: get_entries_count()	Retrieves the specified number of records in a module.
Call: get_report_entries()	Retrieves a list of report entries based on specified report IDs.

Call: get_entry()
Retrieves a single SugarBean based on ID.
Syntax

get_entry(session, module_name, id,select_fields, link_name_to_fields_array)

Arguments

Name	Type	Description
session	String	Session ID returned by a previous login call.
module_name	String	The name of the module from which to retrieve records. Note: If you change the module's tab name in Studio, it does not affect the name that must be passed into this method.
id	String	The SugarBean's ID.
select_fields	Array	Optional. The list of fields to be returned in the results.
link_name_to_fields_array	Array	A list of link names and the fields to be returned for each link name.



Example: 'link_name_to_fields_array' =>
 array(array('name' => 'email_addresses', 'value'
 => array('id', 'email_address', 'opt_out',
 'primary_address')))

Output

Name	Type	Description
entry_list	Array	The record's name-value pair for the simple datatypes excluding the link field data.
relationship_list	Array	The records link field data.

Call: `get_entries()`

Retrieves a list of SugarBeans based on the specified IDs.

Syntax

`get_entries(session, module_name, ids, select_fields, link_name_to_fields_array)`

Arguments

Name	Type	Description
session	String	Session ID returned by a previous login call.
module_name	String	The name of the module from which to retrieve records. Note: If you change the module's tab name in Studio, it does not affect the name that must be passed into this method.
ids	Array	An array of SugarBean IDs.
select_fields	Array	Optional. The list of fields to be returned in the results.
link_name_to_fields_array	Array	A list of link names and the fields to be returned for each link name. Example: 'link_name_to_fields_array' => array(array('name' => 'email_addresses', 'value' => array('id', 'email_address', 'opt_out', 'primary_address')))

Output

Name	Type	Description
------	------	-------------



entry_list	Array	The record's name-value pair for the simple datatypes excluding the link field data.
relationship_list	Array	The records link field data.

Call: `get_entry_list()`

Retrieves a list of SugarBeans.

Syntax

`get_entry_list(session, module_name, query, $order_by, offset, select_fields, link_name_to_fields_array, max_results, deleted)`

Arguments

Name	Type	Description
session	String	Session ID returned by a previous login call.
module_name	String	The name of the module from which to retrieve records. Note: If you change the module's tab name in Studio, it does not affect the name that must be passed into this method.
query	String	The SQL WHERE clause without the word "where".
order_by	String	The SQL ORDER BY clause without the phrase "order by".
offset	String	The record offset from which to start.
select_fields	Array	Optional. A list of fields to include in the results.
link_name_to_fields_array	Array	A list of link names and the fields to be returned for each link name. Example: 'link_name_to_fields_array' => array(array('name' => 'email_addresses', 'value' => array('id', 'email_address', 'opt_out', 'primary_address')))
max_results	String	The maximum number of results to return.
deleted	Number	To exclude deleted records

Output



Name	Type	Description
result_count	Integer	The number of returned records.
next_offset	Integer	The start of the next page.
entry_list	Array	The records that were retrieved.
relationship_list	Array	The records' link field data.

Call: `set_relationship()`

Sets a single relationship between two SugarBeans.

Syntax

`set_relationship(session, module_name, module_id, link_field_name, related_ids)`

Arguments

Name	Type	Description
session	String	Session ID returned by a previous login call.
module_name	String	The name of the module from which to retrieve records. Note: If you change the module's tab name in Studio, it does not affect the name that must be passed into this method.
module_id	String	The ID of the specified module bean.
link_field_name	String	The name of the field related to the other module.
related_ids	Array	IDs of related records

Output

Name	Type	Description
created	Integer	The number of relationships that were created.
failed	Integer	The number of relationships that failed.
deleted	Integer	The number of relationships that were deleted.

Call: `set_relationships()`

Sets multiple relationships between two SugarBeans.

Syntax



set_relationships(session, module_names, module_ids, link_field_names, related_ids)

Arguments

Name	Type	Description
session	String	Session ID returned by a previous login call.
module_names	Array	The name of the module from which to retrieve records. Note: If you change the module's tab name in Studio, it does not affect the name that must be passed into this method.
module_ids	Array	The ID of the specified module bean.
link_field_names	Array	The name of the field related to the other module.
related_id	Array	IDs of related records

Output

Name	Type	Description
created	Integer	The number of relationships that were created.
failed	Integer	The number of relationships that failed.
deleted	Integer	The number of relationships that were deleted.

Call: get_relationship()

Retrieves a collection of beans that are related to the specified bean and, optionally, returns relationship data.

Syntax

get_relationships(session, module_name, module_id, link_field_name, related_module_query, related_fields, related_module_link_name_to_fields_array, deleted)

Arguments

Name	Type	Description
session	String	Session ID returned by a previous login call.
module_name	String	The name of the module from which to retrieve records. Note: If you change the module's tab name in Studio, it does not affect the name that must be passed into this method.



module_ids	String	The ID of the specified module bean.
link_field_name	String	The relationship name of the linked field from which to return records.
related_module_query	String	The portion of the WHERE clause from the SQL statement used to find the related items.
related_fields	Array	The related fields to be returned.
related_module_link_name_to_fields_array	Array	For every related bean returned, specify link field names to field information. Example: 'link_name_to_fields_array' => array(array('name' => 'email_addresses', 'value' => array('id', 'email_address', 'opt_out', 'primary_address')))
deleted	Number	To exclude deleted records

Output

Name	Type	Description
entry_list	Array	The records that were retrieved.
relationship_list	Array	The records' link field data.

Call: set_entry()

Creates or updates a SugarBean.

Syntax

```
set_entry(session,module_name, name_value_list)
```

Arguments

Name	Type	Description
session	String	Session ID returned by a previous login call.
module_name	String	The name of the module from which to retrieve records. Note: If you change the module's tab name in Studio, it does not affect the name that must be passed into this method.
name_value_list	Array	The value of the SugarBean attributes

Output



Name	Type	Description
id	String	The ID of the bean that that was written to.

Call: `set_entries()`

Creates or updates a list of SugarBeans.

Syntax

`set_entries(session,module_name, name_value_lists)`

Arguments

Name	Type	Description
session	String	Session ID returned by a previous login call.
module_name	String	The name of the module from which to retrieve records. Note: If you change the module's tab name in Studio, it does not affect the name that must be passed into this method.
name_value_lists	Array	An array of bean-specific arrays where the keys of the array are SugarBean attributes.

Output

Name	Type	Description
ids	String	The IDs of the beans that that were written to.

Call: `login()`

Logs the user into the Sugar application.

Syntax

`login(user_auth, application)`

Arguments

Name	Type	Description
user_auth	Array	Sets username and password
application	String	Not used. The name of the application from which the user is login in.
name_value_list	Array	Sets the name_value pair. Currently you can use this function to set values for the 'language' and 'notifyonsave' parameters.



The language parameter sets the language for the session. For example, 'name'='language', 'value'='en_US'

The 'notifyonsave' sends a notification to the assigned user when a record is saved. For example, 'name'='notifyonsave', 'value'='true'.

Output

Name	Type	Description
id	String	The session ID
module_name	String	The Users module
name_value_list	Array	The name-value pair of user ID, user name, and user language, user currency ID, and user currency name.

Call: `logout()`

Logs out of the Sugar user session.

Syntax

`logout($session)`

Example:

To log out a user:

```
logout array('user_auth' =>
array('session'=>' 4d8d6c12a0c519a6eff6171762ac252c')
```

Arguments

Name	Type	Description
session	String	Session ID returned by a previous call to login.

This call does not return an output.

Call: `get_server_info()`

Returns server information such as version, flavor, and gmt_time.

Syntax

`get_server_info()`

Arguments

Name	Type	Description
None	Null	No Arguments

Output



Name	Type	Description
flavor	String	Sugar edition such as Enterprise, Professional, or Community Edition.
version	String	The version number of the Sugar application that is running on the server.
gmt_time	String	The current GMT time on the server in Y-m-d H:i:s format.

Call: `get_user_id()`

Returns the ID of the user who is logged into the current session.

Syntax

```
new_get_user_id array('session' => sessionid)
```

Arguments

Name	Type	Description
session	String	Returns the User ID of the current session

Output

Name	Type	Description
user id	String	The user ID

Call: `get_module_fields()`

Retrieves variable definitions (vardefs) for fields of the specified SugarBean.

Syntax

```
get_module_fields(session, module_name, fields)
```

Arguments

Name	Type	Description
session	String	Returns the session ID
module_name	String	The module from which to retrieve vardefs.
fields	Array	Optional. Returns vardefs for the specified fields.

Output

Name	Type	Description
------	------	-------------



module_fields	Array	The vardefs of the module fields.
link_fields	Array	The vardefs for the link fields.

Call: seamless_login()

Performs a seamless login during synchronization.

Syntax

seamless_login(session)

Arguments

Name	Type	Description
session	String	Returns the session ID

Output

Name	Type	Description
1	Integer	If the session is authenticated
0	Integer	If the session is not authenticated

Call: set_note_attachment()

Add or replace a note's attachment. Optionally, you can set the relationship of this note to related modules using related_module_id and related_module_name.

Syntax

set_note_attachment(session, note)

Arguments

Name	Type	Description
session	String	The session ID returned by a previous call to login.
note	Array	The ID of the note containing the attachment.
filename	String	The file name of the attachment.
file	Binary	The binary contents of the file.
related_module_id	String	The id of the module to which this note is related.
related_module_name	String	The name of the module to which this note is related.



Output

Name	Type	Description
id	String	The ID of the note.

get_note_attachment()

Retrieves an attachment from a note.

Syntax

```
get_note_attachment(session,id)
```

Arguments

Name	Type	Description
session	String	The ID of the session
id	String	The ID of the note.

Output

Name	Type	Description
id	String	The ID of the note containing the attachment.
filename	String	The file name of the attachment.
file	Binary	The binary contents of the file.
related_module_id	String	The id of the module to which this note is related.
related_module_name	String	The name of the module to which this note is related.

Call: set_document_revision()

Sets a new revision for a document.

Syntax

```
set_document_revision(session, document_revision)
```

Arguments

Name	Type	Description
session	String	Returns the session ID
document_revision	String	The document ID, document name, the revision number, the file name of the attachment, the binary contents of the file.



id	String	The document revision ID.
----	--------	---------------------------

Output

Name	Type	Description
id	String	The ID of the document revision.

Call: `get_document_revision()`

In case of .htaccess lock-down on the cache directory, allows an authenticated user with the appropriate permissions to download a document.

Syntax

`get_document_revision(session, id)`

Arguments

Name	Type	Description
session	String	Returns the session ID
id	String	The ID of the revised document

Output

Name	Type	Description
document_revision_id	String	The ID of the document revision containing the attachment.
document_name	String	The name of the revised document
revision	String	The revision value
filename	String	The file name of the attachment
file	Binary	The binary contents of the file.

Call: `search_by_module()`

Returns the ID, module name and fields for specified modules. Supported modules are Accounts, Bugs, Calls, Cases, Contacts, Leads, Opportunities, Projects, Project Tasks, and Quotes.

Syntax

`search_by_module(session, search_string, modules, offset, max_results)`

Arguments

Name	Type	Description
------	------	-------------



session	String	The session ID returned by a previous call to login
search_string	String	The string to search for
modules	String	The modules to query
offset	Integer	The specified offset in the query
max_results	Integer	The maximum number of records to return

Output

Name	Type	Description
return_search_result	Array	The records returned by the search results.

Call: `get_available_modules()`

Retrieves the list of modules available to the current user logged into the system.

Syntax

`get_available_modules (session)`

Arguments

Name	Type	Description
session	String	The session ID returned by a previous call to login

Output

Name	Type	Description
modules	Array	An array of available modules

Call: `get_user_team_id()`

Retrieves the ID of the default team of the user who is logged into the current session.

Syntax

`get_user_team_id (session)`

Arguments

Name	Type	Description
session	String	The session ID returned by a previous call to login

Output



Name	Type	Description
team_id	String	The ID of the current user's default team.

Call: `set_campaign_merge()`

Performs a mail merge for the specified campaign.

Syntax

`set_campaign_merge (session,targets,campaign_id)`

Arguments

Name	Type	Description
session	String	The session ID returned by a previous call to login
targets	Array	A string array of IDs identifying the targets used in the merge.
campaign-id	String	The campaign ID used for the mail merge.

This call does not return an output.

Call: `get_entries_count()`

Retrieves the specified number of records in a module.

Syntax

`get_entries_count(session, module_name,query, deleted)`

Arguments

Name	Type	Description
session	String	The session ID returned by a previous call to login
module_name	String	The name of the module from which to retrieve the records
query	String	Allows the webservice user to provide a WHERE clause.
deleted	Integer	Specifies whether or not to include deleted records.

Output

Name	Type	Description
------	------	-------------



result_count	Integer	Total number of records for the specified query and module
--------------	---------	--

Call: `get_report_entries()`

(Sugar Enterprise and Professional only)

Retrieves a list of report entries based on specified report IDs.

Syntax

`get_report_entries(session,ids,select_fields)`

Arguments

Name	Type	Description
session	String	The session ID returned by a previous call to login
ids	Array	The array of report IDs.
select_fields	String	Optional. The list of fields to be included in the results. This parameter enables you to retrieve only required fields.

Output

Name	Type	Description
field_list	String	Vardef information about the returned fields.
entry_list	String	Array of retrieved records.

Sample Code

```
require_once('include/nusoap/nusoap.php');
$soapClient = new nusoapclient('http://YOURSUGARINSTANCE/service/v2/soap.php?wsdl',true);
$userAuth = $soapClient->call('login',
    array('user_auth' =>
        array('user_name' => 'jim',
            'password' => md5('jim'),
            'version' => '.01',
            'application_name' => 'SoapTest')));
$sessionId = $userAuth['id'];
$reportIds = array('id'=>'c42c1789-c8a6-5876-93a3-4c1ff15d17f6');
$myReportData = $soapClient->call('get_report_entries',
    array('session'=>$sessionId,
        'ids'=>$reportIds));
```

Sample Request For User Login

Sample Request



```

<soapenv:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:soapenv="http://schemas.xmlsoap.org/soap/
envelope/" xmlns:sug="http://www.sugarcrm.com/sugarcrm">
<soapenv:Header/>
<soapenv:Body>
<sug:login soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<user_auth xsi:type="sug:user_auth">
<!--You may enter the following 2 items in any order-->
<user_name xsi:type="xsd:string">admin</user_name>
<password xsi:type="xsd:string">0cc175b9c0f1b6a831c399e269772661
</password>
</user_auth>
<application_name xsi:type="xsd:string"/>
</sug:login>
</soapenv:Body>
</soapenv:Envelope>

```

Sample Response

```

<SOAP-ENV:Envelope SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/" xmlns:xsd="http://www.w3.org/2001/
/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:SOAP-
ENC="http://schemas.xmlsoap.org/soap/encoding/" xmlns:tns="http://www.sugarcrm.com/sugarcrm">
<SOAP-ENV:Body>
<ns1:loginResponse xmlns:ns1="http://www.sugarcrm.com/sugarcrm">
<return xsi:type="tns:entry_value">
<id xsi:type="xsd:string">5b7f9c396370d1116affa5f863695c60</id>
<module_name xsi:type="xsd:string">Users</module_name>
<name_value_list xsi:type="SOAP-ENC:Array" SOAP-ENC:arrayType="tns:name_value[5]">
<item xsi:type="tns:name_value">
<name xsi:type="xsd:string">user_id</name>
<value xsi:type="xsd:string">1</value>
</item>
<item xsi:type="tns:name_value">
<name xsi:type="xsd:string">user_name</name>
<value xsi:type="xsd:string">admin</value>
</item>
<item xsi:type="tns:name_value">
<name xsi:type="xsd:string">user_language</name>
<value xsi:type="xsd:string">en_us</value>
</item>
<item xsi:type="tns:name_value">
<name xsi:type="xsd:string">user_currency_id</name>
<value xsi:nil="true" xsi:type="xsd:string"/>
</item>
<item xsi:type="tns:name_value">
<name xsi:type="xsd:string">user_currency_name</name>
<value xsi:type="xsd:string">US Dollars</value>
</item>
</name_value_list>
</return>
</ns1:loginResponse>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>

```



Extensibility in Upgrade Safe Manner

As of version 5.5.0, Sugar provides versioning for upgrade safe extensibility (Please refer to the Versioning section for details).

Follow the steps outlined below to create your own upgrade-safe version.

- 1) The services directory contains a v2 directory. Create a v2_1 directory in the custom directory. You can create this directory anywhere in the directory structure of source code but it is best to put it in the custom directory so that it is upgrade safe.
- 2) You need to provide your own registry.php. For example, customregistry.php, and the source code looks like this:

```
<?php
require_once('service/v2/registry.php');
class customregistry extends registry{
public function __construct($serviceClass) {
parent::__construct($serviceClass);
} // fn
protected function registerFunction() {
parent::registerFunction();
$this->serviceClass->registerFunction(
'get_entry',
array('session'=>'xsd:string', 'module_name'=>'xsd:string',
'id'=>'xsd:string'),
array('return'=>'xsd:string'));
} // fn
} // class
```

- 3) Look at the registerFunction. We call parent:: registerFunction() to include all the out of box functions and the next line is \$this->serviceClass->registerFunction(name, input, output). This allows you to provide your own functions.
- 4) You need to provide your own implementation class which extends from the base implementation class. For example.

```
<?php
require_once('service/core/SugarWebServiceImpl.php');
class SugarWebServiceImpl_v2_1 extends SugarWebServiceImpl {
function get_entry($session, $module_name, $id){
return $id;
} // fn
} // class
```

- 5) You need to provide your own soap.php or rest.php and initialize all the variables.

The following is an example of your own soap.php

```
<?php
Chdir('../') (based on how much deep you have defines v2_1 directory)
```



```
$webservice_class = 'SugarSoapService2';
$webservice_path = 'service/v2/SugarSoapService2.php';
$registry_class = 'customregistry ';
$registry_path = 'service/v2_1/customregistry.php';
$webservice_impl_class = 'SugarWebServiceImpl_v2_1';
$location = '/service/v2_1/soap.php';
require_once('../core/webservice.php');
?>
```

Now your new SOAP URL will be http://localhost/service/v2_1/soap.php.

Following is an example of your rest.php

```
<?php
Chdir('../') (based on how much deep you have defines v2_1 directory)
$webservice_class = 'SugarRestService2';
$webservice_path = 'service/v2/SugarRestService2.php';
$registry_class = 'customregistry ';
$registry_path = 'service/v2_1/customregistry.php';
$webservice_impl_class = 'SugarRestServiceImpl_v2_1';
$location = '/service/v2_1/rest.php';
require_once('../core/webservice.php');
?>
```

Now your new REST URL will be http://localhost/service/v2_1/rest.php. Your v2_1 directory is now upgrade safe.

SOAP Errors

We will set the fault object on server in case of any exception. So the client has to check for errors and call the appropriate method to get the error code and description from that object

Support WS-I 1.0 Basic profile for WSDL

Sugar has provided support to generate a URL that is WS-I compliant. to access the WSDL file in the format `http://<hostname>/service/v2/soap.php?wsdl`.

Hence, the new URL will look like this: `http://<hostname>/service/v2/soap.php?wsdl&style=rpc&use=literal`.

The style parameter can take either 'rpc' or 'document' and use parameter can be 'literal' or 'encoded'. If you don't specify style and use parameter then default will be rpc/encoded.

This WSDL (rpc/literal) was successfully tested with Apache CXF 2.2.

SugarSoap Examples

See `examples/SoapFullTest_Version2.php` for soap examples.

Note: it is also possible to create a NuSOAP client as follows without requiring the WSDL:
`$ soapclient = new nusoapclient('http://www.example.com/sugar/soap.php',false);`



This speeds up processing because downloading the WSDL before invoking the method is time intensive. This type of URL (<http://www.example.com/sugar/soap.php>) without havin ?wsdl at the end work fine with NUSOAP client. But with clients like .net, java you need to generate all the client side classes for all the complex type data types by giving appropriate WSDL (rpc or document and literal and encoded) and make a service call by coding it to those generated classes.

Cloud Connectors Framework

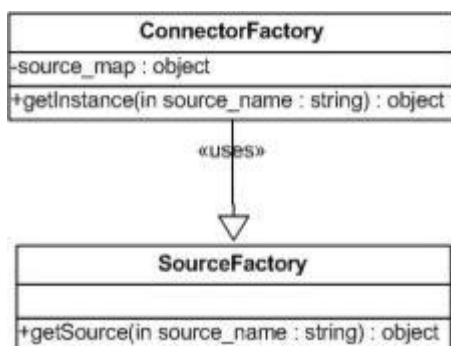
This section covers the design specifications for the connector integration capabilities in SugarCRM called "Cloud Connectors". The Cloud Connector framework allows for various data retrieved through REST and SOAP protocols to be easily viewed and entered into SugarCRM.

The Cloud Connector framework is designed to provide an abstract layer around a connector. So, essentially our own database would just be considered another connector along with any Soap or, REST connector. These connectors, in theory, can then be swapped in and out seamlessly. Hence, providing the framework for it and leveraging a small component is a step in the right direction. The connectors can then be loaded in by a factory, returned, and called based on their interface methods.

The main components for the connector framework are the factories, source, and formatter classes. The factories are responsible for returning the appropriate source or formatter instance for a connector. The sources are responsible for encapsulating the retrieval of the data as a single record or a list or records of the connectors. The formatters are responsible for rendering the display elements of the connectors.

Factories

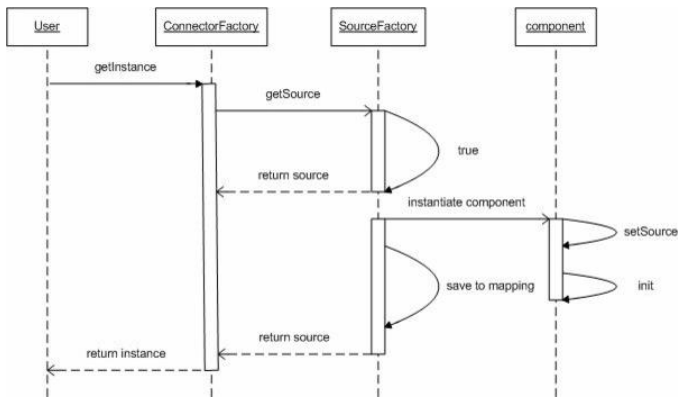
The primary factory is the ConnectorFactory class. It uses the static SourceFactory class to return a connector instance.



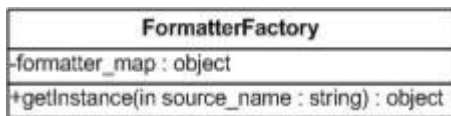
The main points to note are that given a source name (e.g. "ext_soap_hoovers"), the underscore characters are replaced with the file separator character. In this case, "ext_soap_hoovers" becomes "ext/soap/hoovers". Then the SourceFactory first scans the modules/Connectors/connectors/sources and then the custom/modules/Connectors/connectors/sources directories to check for the presence of this



file, and returns a source instance if the file is found. The following sequence diagram attempts to highlight the aforementioned steps.



There is also the FormatterFactory class to return a formatter instance. Retrieving a formatter instance is similar to retrieving a source instance except that the FormatterFactory scans in order the modules/Connectors/connectors/formatters first and then the custom/modules/Connectors/connectors/formatters directories.



Sources

The sources are the centerpiece of the Connectors framework. There are two categories of sources- REST implementations and SOAP implementations. The default source class is abstract and subclasses need to override the getList and getItem methods. The class name of the source should be prefixed with either "ext_soap_" or "ext_rest_". This is because the "_" character serves as a delimiter into the file system for the class to be found. For example, a SOAP implementation for a source we call "Test" will have the class name "ext_soap_test" and a REST implementation will have the class name "ext_rest_test".

```

/**
 * getItem
 * Returns an array containing a key/value pair(s) of a source record
 *
 * @param Array $args Array of arguments to search/filter by
 * @param String $module String optional value of the module that the connector framework is
 attempting to map to
 * @return Array of key/value pair(s) of the source record; empty Array if no results are found
 */
public function getItem($args=array(), $module=null){}

/**
 * getList
 * Returns a nested array containing a key/value pair(s) of a source record
 *
 * @param Array $args Array of arguments to search/filter by
  
```



```

* @param String $module String optional value of the module that the connector framework is
attempting to map to
* @return Array of key/value pair(s) of source record; empty Array if no results are found
*/
public function getList($args=array(), $module=null){}

```

Here is an example of the Array format for the getItem method of the Test source:

```

Array(
['id'] => 19303193202,
['duns'] => 19303193202,
['rename'] => 'SugarCRM, Inc',
['addrcity'] => 'Cupertino',
)

```

Here is an example of the Array format for the getList method of the Test source:

```

Array(
[19303193202] => Array(
['id'] => 19303193202,
['duns'] => 19303193202,
['rename'] => 'SugarCRM, Inc',
['addrcity'] => 'Cupertino',
),
[39203032990] => Array(
['id'] => 39203032990,
['duns'] => 39203032990,
['rename'] => 'Google',
['addrcity'] => 'Mountain View',
)
)

```

The key values for the getList/getItem entries should be mapped to a vardefs.php file contained with the source. This vardefs.php file is required. In this case, we have something like:

```

<?php
$dictionary['ext_rest_test'] = array(
'comment' => 'vardefs for test connector',
'fields' => array (
'id' => array (
'name' => 'id',
'vname' => 'LBL_ID',
'type' => 'id',
'hidden' => true
'comment' => 'Unique identifier'
),
'addrcity' => array (
'name' => 'addrcity',
'input' => 'bal.location.city',
'vname' => 'LBL_CITY',
'type' => 'varchar',
'comment' => 'The city address for the company',
'options' => 'addrcity_dom',
'search' => true,
),
)
)

```



```
);  
?>
```

Note the 'input' key for the addrcity entry. The 'input' key allows for some internal argument mapping conversion that the source uses. The period (.) is used to delimit the input value into an Array. In the example of the addrcity entry, the value bal.location.city will be translated into the Array argument ['bal']['location']['city'].

The 'search' key for the addrcity entry may be used for the search form in the Connectors' data merge wizard screens available for the professional and enterprise editions.

Finally, note the 'options' key for the addrcity entry. This 'options' key maps to an entry in the mapping.php file to assist in the conversion of source values to the database value format in SugarCRM. For example, assume a source that returns American city values as a numerical value (San Jose = 001, San Francisco = 032, etc.). Internally, the SugarCRM system may use the city airport codes (San Jose = sjc, San Francisco = sfo). To allow the connector framework to map the values, the options configuration is used.

Sources also need to provide a config.php file that may contain optional runtime properties such as the URL of the SOAP WSDL file, API keys, etc. These runtime properties shall be placed under the 'properties' Array. At a minimum, a 'name' key should be provided for the source.

```
<?php  
$config = array (  
'name' => 'Test', //Name of the source  
'properties' =>  
array (  
'TEST_ENDPOINT' => 'http://test-dev.com/axis2/services/AccessTest',  
'TEST_WSDL' => 'http://hapi-dev.test.com/axis2/test.wsdl',  
'TEST_API_KEY' => 'abc123',  
),  
);  
?>
```

An optional mapping.php file may be provided so that default mappings are defined. These mappings assist the connector framework's component class. In the component class there are the fillBean and fillBeans methods. These methods use the getItem/getList results from the source instance respectively and return a SugarBean/SugarBeans that map the resulting values from the source to fields of the SugarBean(s). While the mapping.php file is optional, it should be created if the 'options' key is used in vardefs entries in the vardefs.php file.

```
<?php  
$mapping = array (  
'beans' => array (  
'Leads' => array (  
'id' => 'id',  
'addrcity' => 'primary_address_city',  
),  
),  
);
```



```

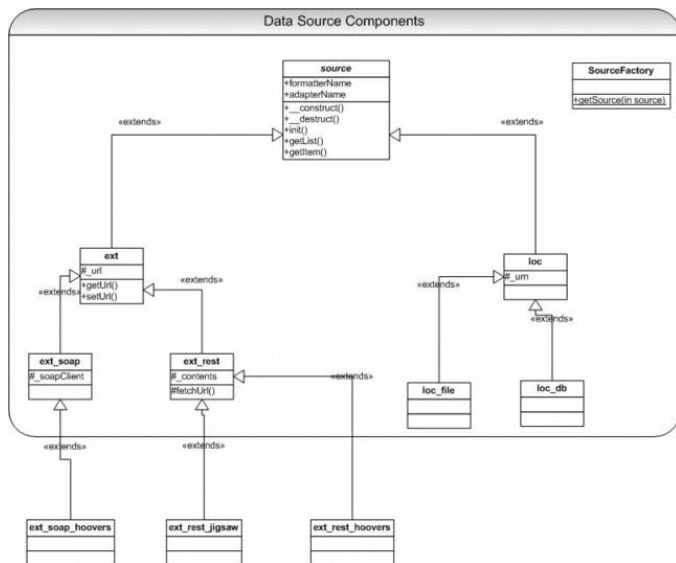
'Accounts' => array (
'id' => 'id',
'addrcity' => 'primary_address_city',
),
),
'options' => array (
'addrcity_dom' => array (
'001' => 'sjc', //San Jose
'032' => 'sfo', //San Francisco
),
),
);
?>

```

In this example, there are two modules that are mapped (Leads and Accounts). The source keys are the Array keys while the SugarCRM module's fields are the values. Also note the example of the 'options' Array as discussed in the vardefs.php file section. Here we have defined an addrcity_dom element to map numerical city values to airport codes.

The source file (test.php) and its supporting files will be placed into self contained directories. In our test example, the contents would be as follows:

- *custom/modules/Connectors/connectors/sources/ext/rest/test/test.php
- *custom/modules/Connectors/connectors/sources/ext/rest/test/vardefs.php
- *custom/modules/Connectors/connectors/sources/ext/rest/test/config.php
- *custom/modules/Connectors/connectors/sources/ext/rest/test/mapping.php



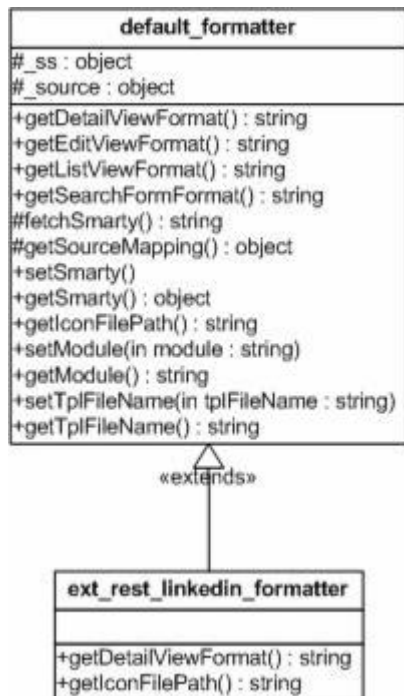
Formatters

The optional formatter components are used by the connector framework to render a popup window that may display additional details and information. Currently, they are shown in the detail view screens for modules that are enabled for the extends connector. Like the source class, the formatter class has a



corresponding factory class (FormatterFactory). The formatters also follow the same convention of using the "ext_rest_" or "ext_soap_" prefix. However, to distinguish conflicting class names, a suffix "_formatter" is also used. Formatters extend from default_formatter.

The following class diagram shows an example of the LinkedIn formatter extending from the default formatter.



Here, we have a subclass ext_rest_linkedin_formatter that overrides the getDetailViewFormat and getIconFilePath methods.

```
require_once('include/connectors/formatters/default/formatter.php');
```

```
class ext_rest_linkedin_formatter extends default_formatter {
public function getDetailViewFormat() {
    $mapping = $this->getSourceMapping();
    $mapping_name = !empty($mapping['beans'][$this->_module]['name']) ? $mapping['beans'][$this->_module]['name'] : "";

```

```
if(!empty($mapping_name)) {
    $this->_ss->assign('mapping_name', $mapping_name);
    return $this->fetchSmarty();
}

```

```
$GLOBALS['log']->error($GLOBALS['app_strings']['ERR_MISSING_MAPPING_ENTRY_FORM_MODULE']);
return "";
}

```

```
public function getIconFilePath() {
return 'modules/Connectors/connectors/formatters/ext/rest/linkedin/tpls/linkedin.gif';
}

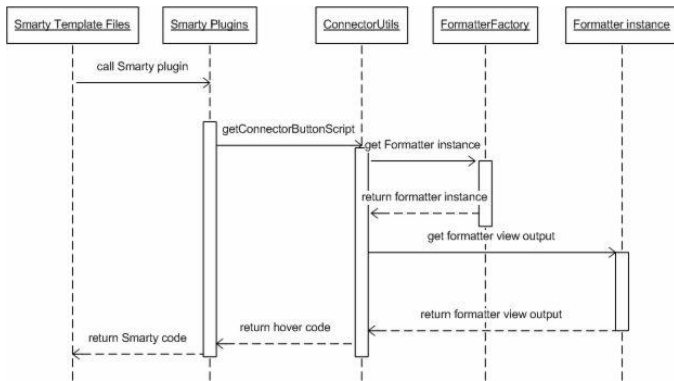
```



}

The `default_formatter` class provides an implementation of the `getDetailViewFormat` method. This method is responsible for rendering the hover code that appears next to certain Detail View fields. The `default_formatter` class will scan the `tpls` directory for a Smarty template file named after the module that is being viewed. For example, the file `*formatters/ext/rest/linkedin/tpls/Accounts.tpl` will be used for the Accounts popup view if the file exists. If the module named template file is not found, it will attempt to use a file named `default.tpl`.

Formatters are invoked from the Smarty template code, which in turn uses Smarty plugins to call the Formatter classes. The following sequence diagram illustrates this.



Copyright 2004-2010 SugarCRM Inc.
[Community Edition License](#)



Module Framework

1. [Overview](#)
2. [User Interface Framework](#)
 - 2.1. [Model-View-Controller \(MVC\)](#)
 - 2.2. [SugarCRM MVC Implementation](#)
 - 2.3. [Model](#)
 - 2.3.1. [Sugar Object Templates](#)
 - 2.3.2. [File Structure](#)
 - 2.3.3. [Implementation](#)
 - 2.3.4. [Performance Considerations](#)
 - 2.3.4.1. [Cache Files](#)
 - 2.4. [Controller](#)
 - 2.5. [Upgrade-Safe Implementation](#)
 - 2.5.1. [File Structure](#)
 - 2.5.2. [Implementation](#)
 - 2.6. [Mapping actions to files](#)
 - 2.7. [Upgrade-Safe Implementation](#)
 - 2.7.1. [File Structure](#)
 - 2.7.2. [Implementation](#)
 - 2.8. [Classic Support \(Not Recommended\)](#)
 - 2.8.1. [File Structure](#)
 - 2.9. [Controller Flow Overview](#)
 - 2.10. [View](#)
 - 2.10.1. [Methods](#)
 - 2.11. [Loading the View](#)
 - 2.11.1. [Implementation](#)
 - 2.11.2. [File Structure](#)
 - 2.11.3. [_](#)
 - 2.12. [Display Options for Views](#)
 - 2.12.1. [Implementation](#)
3. [Metadata Framework](#)
 - 3.1. [Background](#)
 - 3.2. [Application Metadata](#)
 - 3.3. [Module Metadata](#)
 - 3.4. [SearchForm Metadata](#)



- 3.5. [DetailView and EditView Metadata](#)
- 3.6. [SugarField Widgets](#)
 - 3.6.1. [File Structure](#)
 - 3.6.2. [Implementation](#)
 - 3.6.3. [SugarFields Widgets Reference](#)
 - 3.6.4. [Address](#)
- 3.7. [Base](#)
- 3.8. [Bool](#)
- 3.9. [Currency](#)
- 3.10. [Datetime](#)
- 3.11. [Datetimecombo](#)
- 3.12. [Download](#)
- 3.13. [Enum](#)
- 3.14. [File](#)
- 3.15. [Float](#)
- 3.16. [Html](#)
- 3.17. [Image](#)
- 3.18. [Int](#)
- 3.19. [Link](#)
- 3.20. [Multienum](#)
- 3.21. [Phone](#)
 - 3.21.1. [Radioenum](#)
 - 3.21.2. [Readonly](#)
 - 3.21.3. [Relate](#)
- 3.22. [Text](#)
- 3.23. [Username](#)
- 4. [Metadata Framework Summary](#)
 - 4.1. [Formatting Changes](#)
 - 4.2. [Vardefs](#)
 - 4.3. [Dictionary Array](#)
 - 4.4. [Fields Array](#)
 - 4.5. [Indices Array](#)
 - 4.6. [Relationships Array](#)
 - 4.6.1. [Many-to-Many Relationships](#)
 - 4.7. [Subpanels](#)
 - 4.7.1. [One-to-Many Relationships](#)
 - 4.7.2. [Many-to-Many Relationships](#)
 - 4.8. [Relationship Metadata](#)



4.9. [Layout Defs](#)

4.10. [Shortcuts](#)

Overview

A Sugar Module consists of the following files:

- A Vardefs file that specifies the Sugar metadata for the database table, fields, data types, and relationships.

- A SugarBean file that implements the functionality to create, retrieve, update, and delete objects in Sugar. SugarBean is the base class for all business objects in Sugar. Each module implements this base class with additional properties and methods specific to that module.

- Metadata files that define the contents and layout of the Sugar screens.
 - o ListView: lists existing records in the module.
 - o Detail View: displays record details.
 - o EditView: allows user to edit the record.
 - o SubPanels: displays the module's relationship with other Sugar modules.
 - o Popups: displays list of records to link with another record.

User Interface Framework

Model-View-Controller (MVC)

A model-view-controller, or MVC, is a design philosophy that creates a distinct separation between business-logic and display logic.

- **Model** - This is the data object built by the business/application logic needed to present in the user interface. For Sugar, it is represented by the SugarBean and all subclasses of the SugarBean.

- **View** - This is the display layer which is responsible for rendering data from the Model to the end-user.

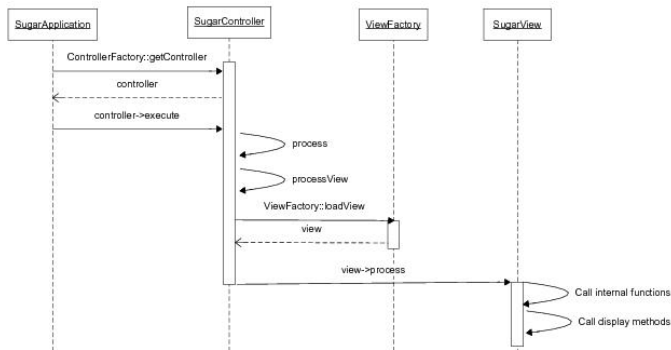


- **Controller** - This is the layer that handles user events such as "Save" and determines what business logic actions to take to build the model, and which view to load for rendering the data to end users.

For more details on the MVC software design pattern, see the [Wikipedia definition](#).

SugarCRM MVC Implementation

The following is a sequence diagram that highlights some of the main components involved within the SugarCRM MVC framework.



Model

The Sugar Model is represented by the SugarBean, and any subclass of the SugarBean. Many of the common Sugar modules also use the SugarObjects class described below.

Sugar Object Templates

Sugar Objects extend the concept of subclassing a step further and allows you to subclass the vardefs. This includes inheriting of fields, relationships, indexes, and language files, but unlike subclassing, you are not limited to a single inheritance. If there were a Sugar Object for fields used across every module such as id, deleted, or date_modified, you could have your module inherit from both Basic Sugar Object and the Person Sugar Object.

For example, the Basic type has a field 'name' with length 10 and Company has a field 'name' with length 20. If you inherit from Basic first then Company your field will be of length 20. Assuming you have defined the field 'name' in your module with length 60, then the module will always override any values provided by Sugar Objects.

There are six types of Sugar Object Templates:

- Basic (contains the basic fields all Sugar modules require)



- Person (used by the Contacts, Prospects and Leads modules)
- Issue (used by the Bugs, Cases modules)
- Company (used by the Accounts module)
- File (based on Documents)
- Sale (based on Opportunities)

We can take this a step further and add *assignable* to the mix. An *assignable* module would be one that can be assigned to users. Although this is not used by every module, many modules do let you assign records to users. SugarObject interfaces allow us to add "assignable" to modules in which we want to enable users to assign records.

SugarObject interfaces and SugarObject templates are very similar to one another, but the main distinction is that templates have a base class you can subclass while interfaces do not. If you look into the file structure you will notice that templates include many additional files including a full metadata directory. This is currently used primarily for Module Builder.

File Structure

- *include/SugarObjects/interfaces*
- *include/SugarObjects/templates*

Implementation

There are two things you need to do to take advantage of SugarObjects:

- 1) Your class needs to subclass the SugarObject class you wish to extend.

```
class MyClass extends Person{
function MyClass(){
parent::Person();
}
}
```

- 2) In your vardefs.php file add the following to the end:

```
VardefManager::createVardef('Contacts','Contact', array('default',
'assignable','team_security', 'person'));
```



This tells the VardefManager to create a cache of the Contacts vardefs with the addition of all the default fields, assignable fields, team security fields (Sugar Professional and Enterprise only), and all fields from the person class.

Performance Considerations

VardefManager caches the generated vardefs into a single file that will be the file loaded at run time. Only if that file is not found will it load the vardefs.php file that is located in your modules directory. The same is true for language files. This caching also includes data for custom fields, and any vardef or language extensions that are dropped into the custom/ext framework.

Cache Files

- `cache/modules/<mymodule>/<object_name>vardefs.php`
- `cache/modules/<mymodule>/languages/en_us.lang.php`

Controller

Version 5.0 introduced a cascading controller concept to increase developer granularity over customizations and to provide additional upgrade-safe capabilities. The main controller, named SugarController, addresses the basic actions of a module from EditView and DetailView to saving a record. Each module can override this SugarController by adding a controller.php file into its directory. This file extends the SugarController, and the naming convention for the class is:
`<ModuleName>Controller`

Inside the controller you define an action method. The naming convention for the method is:
`action_<action_name>`

There are more fine grained control mechanisms that a developer can use to override the controller processing. For example if a developer wanted to create a new Save action there are three places where they could possibly override.

- **action_save** - this is the broadest specification and gives the user full control over the Save process.
- **pre_save** - a user could override the population of parameters from the form
- **post_save** - this is where the view is being setup. At this point the developer could set a redirect url, do some post save processing, or set a different view



Upgrade-Safe Implementation

You can also add a custom Controller that extends the module's Controller if such a Controller already exists. For example, if you want to extend the Controller for a module that comes with Sugar 6.1.0, you should check if that module already has a module-specific controller. If so, you extend from that controller class. Otherwise, you extend from SugarController class. In both cases, you should place the custom controller class file in `custom/modules/<MyModule>/Controller.php` instead of the module directory. Doing so makes your customization *upgrade-safe*.

File Structure

- `include/MVC/Controller/SugarController.php`
- `include/MVC/Controller/ControllerFactory.php`
- `modules/<MyModule>/Controller.php`
- `custom/modules/<MyModule>/controller.php`

Implementation

```
class UsersController extends SugarController{
function action_SetTimeZone(){
//Save TimeZone code in here
...
}
}
```

Mapping actions to files

You can choose not to provide a custom action method as defined above, and instead specify your mappings of actions to files in `$action_file_map`. Take a look at `include/MVC/Controller/action_file_map.php` as an example:

```
$action_file_map['subpanelviewer'] = 'include/SubPanel/SubPanelViewer.php';

$action_file_map['save2'] = 'include/generic/Save2.php';

$action_file_map['deleterelationship'] = 'include/generic/DeleteRelationship.php';

$action_file_map['import'] = 'modules/Import/index.php';
```

Here the developer has the opportunity to map an action to a file. For example Sugar uses a generic



sub-panel file for handling subpanel actions. You can see above that there is an entry mapping the action 'subpanelviewer' to *include/SubPanel/SubPanelViewer.php*.

The base SugarController class loads the action mappings in the following path sequence:

- *include/MVC/Controller*
- *modules/<Module-Name>*
- *custom/modules/<Module-Name>*
- *custom/include/MVC/Controller*

Each one loads and overrides the previous definition if in conflict. You can drop a new *action_file_map* in the later path sequence that extends or overrides the mappings defined in the previous one.

Upgrade-Safe Implementation

If you want to add custom *action_file_map.php* to an existing module that came with the SugarCRM release, you should place the file at *custom/modules/<Module-Name>/action_file_map.php*

File Structure

- *include/MVC/Controller/action_file_map.php*
- *modules/<Module-Name>/action_file_map.php*
- *custom/modules/<Module-Name>/action_file_map.php*

Implementation

```
$action_file_map['soapRetrieve'] = 'custom/SoapRetrieve/soap.php';
```

Classic Support (Not Recommended)

Classic support allows you to have files that represent actions within your module. Essentially, you can drop in a PHP file into your module and have that be handled as an action. This is not recommended, but is considered acceptable for backward compatibility. The better practice is to take advantage of the *action_<myaction>* structure.

File Structure

- *modules/<MyModule>/<MyAction>.php*



Controller Flow Overview

For example, if a request comes in for DetailView the controller will handle the request as follows:

- Start in index.php and load the SugarApplication instance
- SugarApplication instantiates SugarControllerFactory
- SugarControllerFactory loads the appropriate Controller
- Check for custom/modules/<MyModule>/Controller.php
 - o if not found, check for modules/<MyModule>/Controller.php
 - o if not found, load SugarController.php
- Call on the appropriate action
 - o Look for custom/modules/<MyModule>/<MyAction>.php. If found and custom/modules/<MyModule>/views/view.<MyAction>.php is not found, use this view.
 - o If not found check for modules/<MyModule>/<MyAction>.php. If found and modules/<MyModule>/views/view.<MyAction>.php is not found, then use the modules/<MyModule>/<MyAction>.php action
 - o If not found, check for the method action_<MyAction> in the controller.
 - o If not found, check for an action_file_mapping
 - o If not found, report error "Action is not defined"

View

Views display information to the browser. Views are not just limited to HTML data, you can have it send down JSON encoded data as part of the view or any other structure you wish. As with the controllers,



there is a default class called `SugarView` which implements much of the basic logic for views, such as handling of headers and footers.

As a developer, to create a custom view you would place a `view.<view_name>.php` file in a `views/` subdirectory within the module. For example, for the `DetailView`, you would create a file name `view.detail.php` and place this within the `views/` subdirectory within the module. If a `views` subdirectory does not exist, you must create one.

In the file, create a class named: `<Module>View<ViewName>`. For example, for a list view within the `Contacts` module the class would be `ContactsViewList`. Note the first letter of each word is uppercase and all other letters are lowercase.

You can extend the class from `SugarView`, the parent class of all views, or you can extend from an existing view. For example, extending from the out-of-the-box list view can leverage a lot of the logic that has already been created for displaying a list view.

Methods

There are two main methods to be overridden within a view:

- `preDisplay()` - This performs pre-processing within a view. This method is relevant only for extending existing views. For example, the `include/MVC/View/views/view.edit.php` file uses it, and enables developers who wishes to extend this view to leverage all of the logic done in `preDisplay()` and either override the `display()` method completely or within your own `display()` method call `parent::display()`.
- `display()` - This method displays the data to the screen. Place the logic to display output to the screen here.

Loading the View

The `ViewFactory` class tries to load the view for view in this sequence, and will use the first one it finds:

- `custom/modules/<my_module>/views/view.<my_view>.php`
- `modules/<my_module>/views/view.<my_view>.php`
- `custom/include/MVC/View/view.<my_view>.php`
- `include/MVC/Views/view.<my_view>.php`

Implementation

```
class ContactsViewList extends SugarView{
```




```

function ContactsViewList(){
parent::SugarView();
}

function display(){
echo 'This is my Contacts ListView';
}
}

```

File Structure

- *include/MVC/Views/view.<myview>.php*
- *custom/include/MVC/Views/view.<myview>.php*
- *modules/<mymodule>/views/view.<myview>.php*
- *custom/modules/<mymodule>/views/view.<myview>.php*
- *include/MVC/Views/SugarView.php*

Display Options for Views

The Sugar MVC provides developers with granular control over how the screen looks when a view is rendered. Each view can have a config file associated with it. So, in the example above, the developer would place a *view.edit.config.php* within the *views/* subdirectory . When the EditView is rendered, this config file will be picked up. When loading the view, *ViewFactory* class will merge the view config files from the following possible locations with precedence order (high to low):

- *customs/modules/<module-name>/views/view.<my_view>.config.php*
- *modules/<module-name>/views/view.<my_view>.config.php*
- *custom/include/MVC/View/views/view.<my_view>.config.php*
- *include/MVC/View/views/view.<my_view>.config.php*

Implementation

The format of these files is as follows:
`$view_config = array('actions' =>`



```

array('popup' => array(
    'show_header' => false,
    'show_subpanels' => false,
    'show_search' => false, 'show_footer' => false,
    'show_JavaScript' => true,
),
),
'req_params' => array(
    'to_pdf' => array(
        'param_value' => true,
        'show_all' => false
    ),
),
);

```

To illustrate this process, let us take a look at how the 'popup' action is processed. In this case, the system will go to the actions entry within the view_config and determine the proper configuration. If the request contains the parameter *to_pdf*, and is set to be *true* then it will automatically cause the *show_all* configuration parameter to be set *false*, which means none of the options will be displayed.

Metadata Framework

Background

Metadata is defined as information about data. In SugarCRM, metadata refers to the framework of using files to abstract the presentation and business logic found in the system. The metadata framework is described in definition files that are processed using PHP. The processing usually includes the use of Smarty templates for rendering the presentation, and JavaScript libraries to handle some business logic that affects conditional displays, input validation, and so on.

Application Metadata

All application modules are defined in the `modules.php` file. It contains several variables that define which modules are active and usable in the application.

The file is located under the '`<sugar root>/include`' folder. It contains the `$moduleList()` array variable which contains the reference to the array key to look up the string to be used to display the module in the tabs at the top of the application, the coding standard is for the value to be in the plural of the module name; for example, Contacts, Accounts, Widgets, and so on.

The `$beanList()` array stores a list of all active beans (modules) in the application. The `$beanList` entries are stored in a 'name' => 'value' fashion with the 'name' value being in the plural and the 'value' being in



the singular of the module name. The 'value' of a *\$beanList()* entry is used to lookup values in our next modules.php variable, the *\$beanFiles()* array.

The *\$beanFiles* variable is also stored in a 'name' => 'value' fashion. The 'name', typically in singular, is a reference to the class name of the object, which is looked up from the *\$beanList* 'value', and the 'value' is a reference to the class file. From these three arrays you can include the class, instantiate an instance, and execute module functionality.

For example:

```
global $moduleList,$beanList,$beanFiles;
$module_object = 'Contacts';
$class_name = $beanList[$module_object];
$class_file_path = $beanFiles[$class_name];
require_once($class_file_path);
$new_module_object = new $class_name();
$module_string_name = $moduleList[$module_object];
```

The remaining relevant variables in the modules.php file are the *\$modInvisList* variable which makes modules invisible in the regular user interface (i.e., no tab appears for these modules), and the *\$adminOnlyList* which is an extra level of security for modules that are can be accessed only by administrators through the Admin page.

Module Metadata

The following table lists the metadata definition files found in the modules/[module]/metadata directory, and a brief description of their purpose within the system.

File	Description
additionalDetails.php	Used to render the popup information displayed when a user hovers the mouse cursor over a row in the List View.
editviewdefs.php	Used to render a record's EditView.
detailviewdefs.php	Used to render a record's DetailView.
listviewdefs.php	Used to render the List View display for a module.
metafiles.php	Used to override the location of the metadata definition file to be used. The EditView, DetailView, List View, and Popup code check for the presence of these files.
popupdefs.php	Used to render and handle the search form and list view in popups
searchdefs.php	Used to render a module's basic and advanced search form displays



sidecreateviewdefs.php	Used to render a module's quick create form shown in the side shortcut panel
subpaneldefs.php	Used to render a module's subpanels shown when viewing a record's DetailView

SearchForm Metadata

The search form layout for each module is defined in the module's metadata file searchdefs.php. A sample of the Account's searchdefs.php appears as:

```
<?php
$searchdefs['Accounts'] = array(
'templateMeta' => array('maxColumns' => '3',
'widths' => array('label' => '10', 'field' => '30')),
'layout' => array(
'basic_search' => array(
'name',
'billing_address_city',
'phone_office',
array( 'name' => 'address_street',
'label' => 'LBL_BILLING_ADDRESS',
'type' => 'name',
'group'=> 'billing_address_street'
),
array( 'name'=>'current_user_only',
'label'=>'LBL_CURRENT_USER_FILTER',
'type'=>'bool'
),
),
'advanced_search' => array(
'name',
array( 'name' => 'address_street',
'label' =>'LBL_ANY_ADDRESS',
'type' => 'name'
),
array( 'name' => 'phone',
'label' =>'LBL_ANY_PHONE',
'type' => 'name'
),
'website',
array( 'name' => 'address_city',
'label' =>'LBL_CITY',
'type' => 'name'
),
array( 'name' => 'email',
'label' =>'LBL_ANY_EMAIL',
'type' => 'name'
),
),
```



```

'annual_revenue',
array( 'name' => 'address_state',
'label' =>'LBL_STATE',
'type' => 'name'
),
'employees',
array( 'name' => 'address_postalcode',
'label' =>'LBL_POSTAL_CODE',
'type' => 'name'
),
array('name' => 'billing_address_country',
'label' =>'LBL_COUNTRY',
'type' => 'name'
),
'ticker_symbol',
'sic_code',
'rating',
'ownership',
array( 'name' => 'assigned_user_id',
'type' => 'enum',
'label' => 'LBL_ASSIGNED_TO',
'function' => array('name' =>'get_user_array',
'params' => array(false) )
),
'account_type',
'industry',
),
),
);
?>

```

The contents of the `searchdefs.php` file contains the Array variable `$searchDefs` with one entry. The key is the name of the module as defined in `$moduleList` array defined in `include/modules.php`. The value of the `$searchDefs` array is another array that describes the search form layout and fields.

The 'templateMeta' key points to another array that controls the maximum number of columns in each row of the search form ('maxColumns'), as well as layout spacing attributes as defined by 'widths'. In the above example, the generated search form files will allocate 10% of the width spacing to the labels and 30% for each field respectively.

The 'layout' key points to another nested array which defines the fields to display in the basic and advanced search form tabs. Each individual field definition maps to a SugarField widget. See the SugarField widget section for an explanation about SugarField widgets and how they are rendered for the search form, DetailView, and EditView.

The `searchdefs.php` file is invoked from the MVC framework whenever a module's list view is rendered (see `include/MVC/View/views/view.list.php`). Within `view.list.php` checks are made to see if the module has defined a `SearchForm.html` file. If this file exists, the MVC will run in classic mode and use the aforementioned `include/SearchForm/SearchForm.php` file to process the search form. Otherwise, the new search form processing is invoked using `include/SearchForm/SearchForm2.php` and the `searchdefs.php`

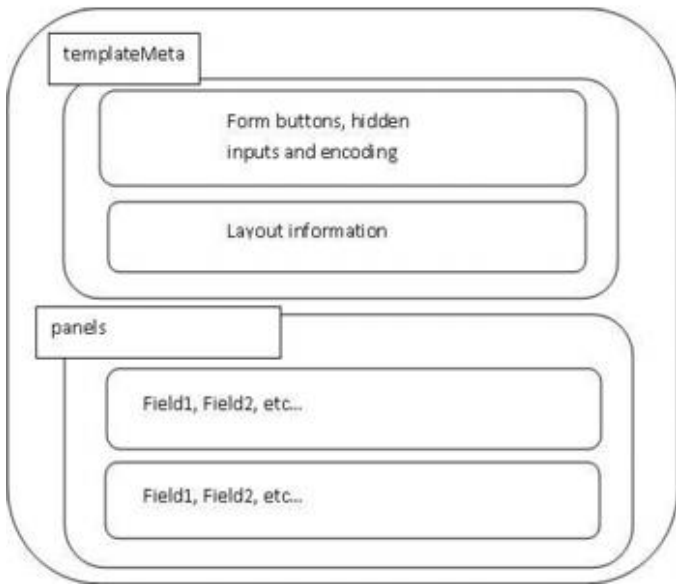


file is scanned for first under the custom/modules/[module]/metadata directory and then in modules/[module]/metadata.

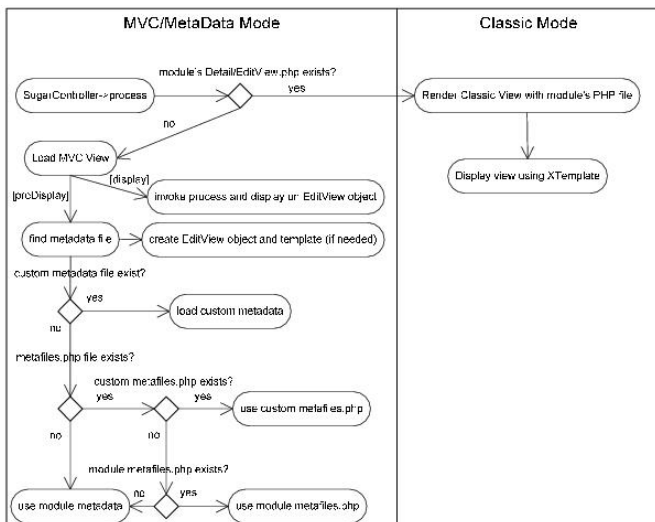
The processing flow for the search form using the metadata subpaneldefs.php file is similar to that of EdiView and DetailView.

DetailView and EditView Metadata

Metadata files are PHP files that declare nested Array values that contain information about the view (buttons, hidden values, field layouts, etc.). A visual diagram that represents how the Array values declared in the Metadata file are nested is as follows:



The following diagram highlights the process of how the application determines which Metadata file is to be used when rendering a request for a view.



The “Classic Mode” on the right hand side of the diagram represents the SugarCRM pre-5.x rendering of a Detail/Editview. This section will focus on the MVC/Metadata mode.

When the view is first requested, the `preDisplay` method will attempt to find the correct Metadata file to use. Typically, the Metadata file will exist in the `[root level]/modules/[module]/metadata` directory, but in the event of edits to a layout through the Studio interface, a new Metadata file will be created and placed in the `[root level]/custom/modules/[module]/metadata` directory. This is done so that changes to layouts may be restored to their original state through Studio, and also to allow changes made to layouts to be upgrade-safe when new patches and upgrades are applied to the application. The `metafiles.php` file that may be loaded allows for the loading of Metadata files with alternate naming conventions or locations. An example of the `metafiles.php` contents can be found for the Accounts module (though it is not used by default in the application).

```
$metafiles['Accounts'] = array(
    'detailviewdefs' => 'modules/Accounts/metadata/detailviewdefs.php',
    'editviewdefs' => 'modules/Accounts/metadata/editviewdefs.php',
    'ListViewdefs' => 'modules/Accounts/metadata/ListViewdefs.php',
    'searchdefs' => 'modules/Accounts/metadata/searchdefs.php',
    'popupdefs' => 'modules/Accounts/metadata/popupdefs.php',
    'searchfields' => 'modules/Accounts/metadata/SearchFields.php',
);
```

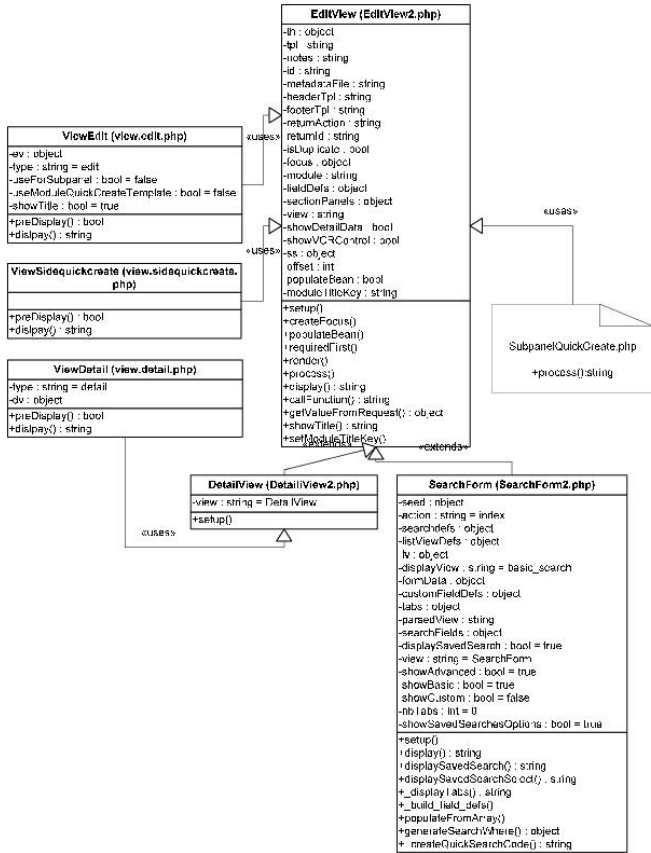
After the Metadata file is loaded, the `preDisplay` method also creates an `EditView` object and checks if a Smarty template file needs to be built for the given Metadata file. The `EditView` object does the bulk of the processing for a given Metadata file (creating the template, setting values, setting field level ACL controls if applicable, etc.). Please see the `EditView` process diagram for more detailed information about these steps.

After the `preDisplay` method is called in the view code, the `display` method is called, resulting in a call to the `EditView` object’s `process` method, as well as the `EditView` object’s `display` method.

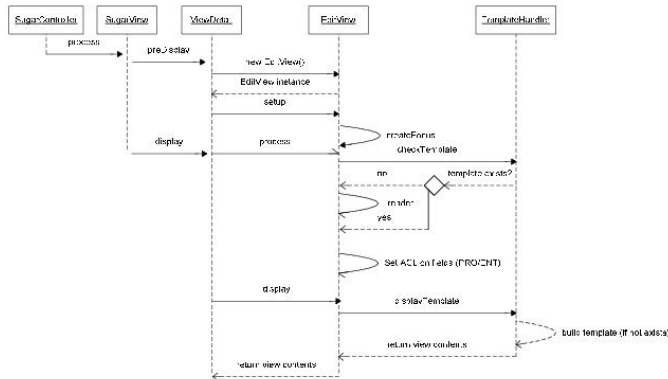
The `EditView` class is responsible for the bulk of the Metadata file processing and creating the resulting display. The `EditView` class also checks to see if the resulting Smarty template is already created. It also applies the field level ACL controls for the Sugar Enterprise and Professional editions.

The classes responsible for displaying the Detail View and SearchForm also extend and use the `EditView` class. The `ViewEdit`, `ViewDetail` and `ViewSidequickcreate` classes use the `EditView` class to process and display its contents. Even the file that renders the quick create form display (`SubpanelQuickCreate.php`) uses the `EditView` class. `DetailView` (in `DetailView2.php`) and `SearchForm` (in `SearchForm2.php`) extend the `EditView` class while `SubpanelQuickCreate.php` uses an instance of the `EditView` class. The following diagram highlights these relationships.





The following diagram highlights the EditView class's main responsibilities and their relationships with other classes in the system. We will use the example of a DetailView request although the sequence will be similar for other views that use the EditView class.

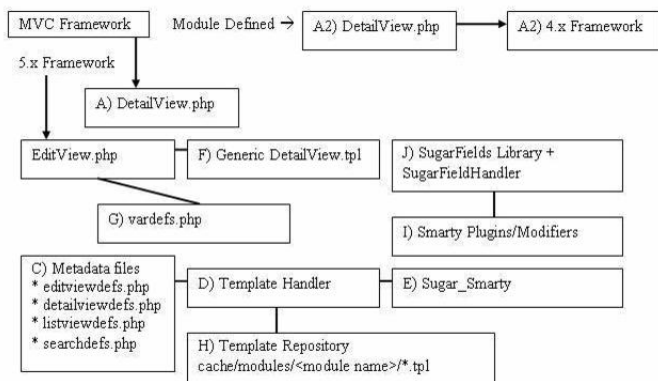


One thing to note is the EditView class's interaction with the TemplateHandler class. The TemplateHandler class is responsible for generating a Smarty template in the cache/modules/<module> directory. For example, for the Accounts module, the TemplateHandler will create the Smarty file cache/modules/Accounts/DetailView.tpl based on the Metadata file definition and other supplementary information from the EditView class. The TemplateHandler class actually uses Smarty itself to generate the resulting template that is placed in the aforementioned cache directory.



Some of the modules that are available in the SugarCRM application also extend the ViewDetail class. One example of this is the DetailView for the Projects module. As mentioned in the MVC section, it is possible to extend the view classes by placing a file in the modules/<module>/views directory. In this case, a view.detail.php file exists in the modules/Projects/views folder. This may serve as a useful example in studying how to extend a view and apply additional field/layout settings not provided by the EditView class.

The following diagram shows the files involved with the DetailView example in more detail.



A high level processing summary of the components for DetailViews follows:

The MVC framework receives a request to process the DetailView.php (A) action for a module. For example, a record is selected from the list view shown on the browser with URL:

index.php?action=DetailView&module=Opportunities&record=46af9843-ccdf-f489-8833

At this point the new MVC framework checks to see if there is a DetailView.php (A2) file in the modules/Opportunity directory that will override the default DetailView.php implementation. The presence of a DetailView.php file will trigger the "classic" MVC view. If there is no DetailView.php (A2) file in the directory, the MVC will also check if you have defined a custom view to handle the DetailView rendering in MVC (that is, checks if there is a file modules/Opportunity/views/view.detail.php). See the documentation for the MVC architecture for more information. Finally, if neither the DetailView.php (A2) nor the view.detail.php exists, then the MVC will invoke include/DetailView/DetailView.php (A).

The MVC framework (see views.detail.php in include/MVC/View/views folder) creates an instance of the generic DetailView (A)

```
// Call DetailView2 constructor
$dv = new DetailView2();
// Assign by reference the Sugar_Smarty object created from MVC
// We have to explicitly assign by reference to back support PHP 4.x
$dv->ss =& $this->ss;
// Call the setup function
$dv->setup($this->module, $this->bean, $metadataFile, 'include/DetailView/DetailView.tpl');
// Process this view
$dv->process();
// Return contents to the buffer
echo $dv->display();
```



When the setup method is invoked, a TemplateHandler instance (D) is created. A check is performed to determine which detailviewdefs.php metadata file to use in creating the resulting DetailView. The first check is performed to see if a metadata file was passed in as a parameter. The second check is performed against the custom/studio/modules/[Module] directory to see if a metadata file exists. For the final option, the DetailView constructor will use the module's default detailviewdefs.php metadata file located under the modules/[Module]/metadata directory. If there is no detailviewdefs.php file in the modules/[Module]/metadata directory, but a DetailView.html exists, then a "best guess" version is created using the metadata parser file in include/SugarFields/Parsers/DetailViewMetaParser.php (not shown in diagram).

The TemplateHandler also handles creating the quick search (Ajax code to do look ahead typing) as well as generating the JavaScript validation rules for the module. Both the quick search and JavaScript code should remain static based on the definitions of the current definition of the metadata file. When fields are added or removed from the file through the Studio application, this template and the resulting updated quick search and JavaScript code will be rebuilt.

It should be noted that the generic DetailView (A) defaults to using the generic DetailView.tpl smarty template file (F). This may also be overridden through the constructor parameters. The generic DetailView (A) constructor also retrieves the record according to the record id and populates the \$focus bean variable.

The *process()* method is invoked on the generic DetailView.php instance:

```
function process() {
//Format fields first
if($this->formatFields) {
$this->focus->format_all_fields();
}
parent::process();
}
```

This in turn, calls the *EditView->process()* method since *DetailView* extends from *EditView*. The *EditView->process()* method will eventually call the *EditView->render()* method to calculate the width spacing for the *DetailView* labels and values. The number of columns and the percentage of width to allocate to each column may be defined in the metadata file. The actual values are rounded as a total percentage of 100%. For example, given the *templateMeta* section's *maxColumns* and *widths* values:

```
'templateMeta' => array('maxColumns' => '2',
'widths' => array(
array('label' => '10', 'field' => '30'),
array('label' => '10', 'field' => '30')
),
),
```



We can see that the labels and fields are mapped as a 1-to-3 ratio. The sum of the widths only equals a total of 80 (10 + 30 x 2) so the actual resulting values written to the Smarty template will be at a percentage ratio of 12.5-to-37.5. The resulting fields defined in the metadata file will be rendered as a table with the column widths as defined:

100%			
50%			
12.5%		37.5%	
Label 1	Value 1	Label 2	Value 2
Label 3	Value 3	Label 4	Value 4
...		...	
...		...	

The actual metadata layout will allow for variable column lengths throughout the displayed table. For example, the metadata portion defined as:

```
'panels' => array (
  'default' => array (
    array (
      'name',
      array (
        'name' => 'amount',
        'label' => '{$MOD.LBL_AMOUNT} ({ $CURRENCY})',
      ),
    ),
    array (
      'account_name',
    ),
    array (
      "",
      'opportunity_type',
    )
  )
)
```

This specifies a default panel under the panels section with three rows. The first row has two fields (name and amount). The amount field has some special formatting using the label override option. The second row contains the account_name field and the third row contains the opportunity_type column.



100%		
50%		
12.5%	37.5%	
Name	Foo	Amount \$100,000
Account	Test Account	
...		Type New Business
...		...

Secondly, the process() method populates the \$fieldDefs array variable with the vardefs.php file (G) definition and the \$focus bean's value. This is done by calling the toArray() method on the \$focus bean instance and combining these value with the field definition specified in the vardefs.php file (G). The display() method is then invoked on the generic DetailView instance for the final step. When the display() method is invoked, variables to the DetailView.tpl Smarty template are assigned and the module's HTML code is sent to the output buffer. Before HTML code is sent back, the TemplateHandler (D) first performs a check to see if an existing DetailView template already exists in the cache repository (H). In this case, it will look for file cache/modules/Opportunity/DetailView.tpl. The operation of creating the Smarty template is expensive so this operation ensures that the work will not have to be redone. As a side note, edits made to the DetailView or EditView through the Studio application will clear the cache file and force the template to be rewritten so that the new changes are reflected. If the cache file does not exist, the TemplateHandler (D) will create the template file and store it in the cache directory. When the fetch() method is invoked on the Sugar_Smarty class (E) to create the template, the DetailView.tpl file is parsed.

SugarField Widgets

SugarFields are the Objects that render the fields specified in the metadata (for example, your *viewdefs.php files). They can be found in include/SugarFields/Fields. In the directory include/SugarFields/Fields/Base you will see the files for the base templates for rendering a field for DetailView, EditView, ListView, and Search Forms. As well as the base class called SugarFieldBase.

File Structure

- *include/SugarFields/Fields/<fieldname>*
- *include/SugarFields/Fields/<fieldname>/DetailView.tpl*
- *modules/MyModule/vardefs.php*
- *modules/MyModule/metadata/<view>defs.php*



Implementation

This section describes the SugarFields widgets that are found in the include/SugarFields/Fields directory. Inside this folder you will find a set of directories that encapsulate the rendering of a field type (for example, Boolean, Text, Enum, and so on.). That is, there are user interface paradigms associated with a particular field type. For example, a Boolean field type as defined in a module's vardef.php file can be displayed with a checkbox indicating the boolean nature of the field value (on/off, yes/no, 0/1, etc.). Naturally there are some displays in which the rendered user interface components are very specific to the module's logic. In this example, it is likely that custom code was used in the metadata file definition. There are also SugarFields directories for grouped display values (e.g. Address, Datetime, Parent, and Relate).

Any custom code called by the metadata will be passed as unformatted data for numeric entries, and that custom code in the metadata will need individual handle formatting.

SugarFields widgets are rendered from the metadata framework whenever the MVC EditView, DetailView, or ListView actions are invoked for a particular module. Each of the SugarFields will be discussed briefly.

Most SugarFields will contain a set of Smarty files for abstract rendering the field contents and supporting HTML. Some SugarFields will also contain a subclass of SugarFieldBase to override particular methods so as to control additional processing and routing of the corresponding Smarty file to use. The subclass naming convention is defined as:

SugarField[`Sugar Field Type`]

where the first letter of the Sugar Field Type should be in uppercase. The contents should also be placed in a corresponding .php file. For example, the contents of the enum type SugarField (rendered as <select> in HTML) is defined in include/SugarFields/Fields/Enum/SugarFieldEnum. In that file, you can see how the enum type will use one of six Smarty template file depending on the view (edit, detail or search) and whether or not the enum vardef definition has a 'function' attribute defined to invoke a PHP function to render the contents of the field.

SugarFields Widgets Reference

Address

The Address field is responsible for rendering the various fields that together represent an address value. By default SugarCRM renders address values in the United States format:

Street
City, State Zip

The Smarty template layout defined in DetailView.tpl reflects this. Should you wish to customize the layout, depending on the \$current_language global variable, you may need to add new files



`[$current_language].DetailView.tpl` or `[$current_language].EditView.tpl` to the Address directory that reflect the language locale's address formatting.

Within the metadata definition, the Address field can be rendered with the snippet:

```
array (  
  'name' => 'billing_address_street',  
  'hideLabel' => true,  
  'type' => 'address',  
  'displayParams'=>array('key'=>'billing', 'rows'=>2, 'cols'=>30, 'maxlength'=>150)  
)
```

name	The vardefs.php entry to key field off of. Though not 100% ideal, we use the street value
hideLabel	Boolean attribute to hide the label that is rendered by metadata framework. We hide the <code>billing_address_street</code> label because the Address field already comes with labels for the other fields (city, state). Also, if this is not set to false, the layout will look awkward.
type	This is the field type override. <code>billing_address_street</code> is defined as a <code>varchar</code> in the <code>vardefs.php</code> file, but since we are interested in rendering an address field, we are overriding this here
displayParams	key - This is the prefix for the address fields. The address field assumes there are <code>[key]_address_street</code> , <code>[key]_address_state</code> , <code>[key]_address_city</code> , <code>[key]_address_postalcode</code> fields so this helps to distinguish in modules where there are two or more addresses (e.g. 'billing' and 'shipping'). rows, cols, maxlength - This overrides the default HTML <code><textarea></code> attributes for the street field component.

Note also the presence of file `include/SugarFields/Fields/Address/SugarFieldAddress.js`. This file is responsible for handling the logic of copying address values (from billing to shipping, primary to alternative, etc.). The JavaScript code makes assumptions using the key value of the grouped fields.

To customize various address formats for different locales, you may provide a locale specific implementation in the folder `include/SugarFields/Fields/Address`. There is a default English implementation provided. Locale implementations are system wide specific (you cannot render an address format for one user with an English locale and another format for another with a Japanese locale). SugarCRM locale settings are system-wide and the SugarField implementation reflects this. To modify based on a user's locale preferences is possible, but will require some customization.

Base

The Base field is the default parent field. It simply renders the value as is for DetailViews, and an HTML text field `<input type="text">` for EditViews. All SugarFields that have a corresponding PHP file extend from `SugarFieldBase.php` or from one of the other SugarFields.



Bool

The Bool field is responsible for rendering a checkbox to reflect the state of the value. All boolean fields are stored as integer values. The Bool field will render a disabled checkbox for DetailViews. If the field value is "1" then the checkbox field will be checked. There is no special parameter to pass into this field from the metadata definition. As with any of the fields you have the option to override the label key string.

For example, in the metadata definition, the Boolean field `do_not_call` can be specified as:

```
'do_not_call'
    or
    array (
        array('name'=>'do_not_call',
            'label'=>'LBL_DO_NOT_CALL' // Overrides label as defined in vardefs.php
        )
    ),
```

Currency

The Currency field is responsible for rendering a field that is formatted according to the user's preferences for currencies. This field's handles will format the field differently if the field name contains the text ``_usd'`, this is used internally to map to the `amount_usdollar` fields which will always display the currency in the user's preferred currency (or the system currency if the user does not have one selected). If the field name does not contain ``_usd'`, the formatting code will attempt to find a field in the same module called ``currency_id'` and use that to figure out what currency symbol to display next to the formatted number. In order for this to work on ListViews and sub-panels, you will need to add the ``currency_id'` column as a ``query_only'` field, for further reference please see the Opportunities ListView definitions.

Within the metadata definition, the Currency field can be rendered with the snippet:

```
'amount', // Assumes that amount is defined
// as a currency field in vardefs.php file
or
array('name'=>'amount',
'displayParams'=>array('required'=>true)
),
```

Datetime

The Datetime field is responsible for rendering an input text field along with an image to invoke the popup calendar picker. The Datetime field differs from the Datetimecombo field in that there is no option to select the time values of a datetime database field.

Within the metadata definition, the Datetime field can be rendered with the snippet:

```
'date_quote_expected_closed', // Assumes that date_quote_expected_closed is defined
// as a datetime field in vardefs.php file
```



```

or
array('name'=>'date_quote_expected_closed',
'displayParams'=>array('required'=>true, 'showFormats'=>true)
),

```

name	Standard name definition when metadata definition is defined as an array
displayParams	required (optional) - Marks the field as required and applies clients side validation to ensure value is present when save operation is invoked from EditView (Overrides the value set in vardefs.php). showFormats (optional) - Displays the user's date display preference as retrieved from the global \$timedate variable's get_user_date_format() method.

Datetimecombo

The Datetimecombo field is similar to the Datetime field with additional support to render dropdown lists for the hours and minutes values, as well as a checkbox to enable/disable the entire field. The date portion (e.g. 12/25/2007) and time portion (e.g. 23:45) of the database fields are consolidated. Hence, the developer must take care to handle input from three HTML field values within the module class code. For example, in the vardefs.php definition for the *date_start* field in the Calls module:

```

'date_start' => array (
  'name' => 'date_start',
  'vname' => 'LBL_DATE',
  'type' => 'datetime',
  'required' => true,
  'comment' => 'Date in which call is schedule to (or did) start'
),

```

There is one database field, but when the Datetimecombo widget is rendered, it will produce three HTML fields for display- a text box for the date portion, and two dropdown lists for the hours and minutes values. The Datetimecombo widget will render the hours and menu dropdown portion in accordance to the user's \$timedate preferences. An optional AM/PM meridiem drop down is also displayed should the user have selected a 12 hour base format (e.g. 11:00).

Within the metadata definition, the Datetimecombo field can be rendered with the snippet:

```

array('name'=>'date_start',
'type'=>'datetimecombo',
'displayParams'=>array('required' => true,
'updateCallback'=>'SugarWidgetScheduler.update_time()';,
'showFormats' => true,
'showNoneCheckbox' => true),
'label'=>'LBL_DATE_TIME'),

```



name	Standard name definition when metadata definition is defined as an array
type	metadata type override. By default, the field defaults to Datetime so we need to override it here in the definition.
displayParams	<p>required (optional) - Marks the field as required and applies clients side validation to ensure value is present when save operation is invoked from EditView (Overrides the value set in vardefs.php).</p> <p>updateCallback (optional) - Defines custom JavaScript function to invoke when the values in the field are changed (date, hours or minutes).</p> <p>showFormats (optional) - Displays the user's date display preference as retrieved from the global \$timedate variable's <code>get_user_date_format()</code> method.</p> <p>showNoneCheckbox (optional) - Displays a checkbox that when checked will disable all three field values</p>
label (optional)	Standard metadata label override just to highlight this exhaustive example

Download

The File field renders a link that references the `download.php` file for the given `displayParam['id']` value when in DetailView mode.

Within the metadata definition, the Download field can be rendered with the snippet:

```
array (
'name' => 'filename',
'displayParams' => array('link'=>'filename', 'id'=>'document_revision_id')
),
```

name	Standard name definition when metadata definition is defined as an array
displayParams	<p>required (optional) - Marks the field as required and applies clients side validation to ensure value is present when save operation is invoked from EditView (Overrides the value set in vardefs.php).</p> <p>id (required for DetailView) - The field for which the id of the file will be opened via the download link.</p> <p>link (required for DetailView) - The field for which the hyperlink value is displayed.</p>



Enum

The Enum field renders an HTML <select> form element that allows for a single value to be chosen. The size attribute of the <select> element is not defined so the element will render as a dropdown field in the EditView.

This field accepts the optional function override behavior that is defined at the vardefs.php file level. For example, in the Bugs module we have for the found_in_release field:

```
'found_in_release'=>
array(
  'name'=>'found_in_release',
  'type' => 'enum',
  'function'=>'getReleaseDropDown',
  'vname' => 'LBL_FOUND_IN_RELEASE',
  'reportable'=>false,
  'merge_filter' => 'enabled',
  'comment' => 'The software or service release that manifested the bug',
  'duplicate_merge' => 'disabled',
  'audited' =>true,
),
```

The function override is not handled by the SugarFields library, but by the rendering code in *include/EditView/EditView2.php*.

Within the metadata definition, the Download field can be rendered with the snippet:

```
array (
  'name' => 'my_enum_field',
  'type' => 'enum',
  'displayParams' => array('javascript'=>'onchange="alert(\'hello world!\');')
),
```

name	Standard name definition when metadata definition is defined as an array
displayParams	size (optional) – Controls the size of the field (affects SearchForm control on the browser only). Defaults to value of 6 for SearchForm. required (optional) - Marks the field as required and applies clients side validation to ensure value is present when save operation is invoked from EditView (Overrides the value set in vardefs.php). javascript (optional) - Custom JavaScript to embed within the <select> tag element (see above example for onchange event implementation).

File

The File field renders a file upload field in EditView, and a hyperlink to invoke download.php in DetailView. Note that you will need to override the HTML form's enctype attribute to be "multipart/form-



data" when using this field and handle the upload of the file contents in your code. This form enctype attribute should be set in the editviewdefs.php file. For example, for the Document's module we have the form override:

```
$viewdefs['Documents']['EditView'] = array(
'templateMeta' => array('form' =>
    array('enctype'=>'multipart/form-data', // <--- override the enctype
    ),
),
```

Within the metadata file, the File field can be rendered with the snippet:

```
array (
'name' => 'filename',
'displayParams' => array('link'=>'filename', 'id'=>'document_revision_id'),
),
```

name	Standard name definition when metadata definition is defined as an array
displayParams	<p>required (optional) - Marks the field as required and applies clients side validation to ensure value is present when save operation is invoked from EditView (Overrides the value set in vardefs.php).</p> <p>id (required for detailviewdefs.php) - The record id that download.php will use to retrieve file contents</p> <p>link (required for detailviewdefs.php) - The text to display between the <a> </> tags.</p> <p>size (optional) - Affects EditView only. Override to set the display length attribute of the input field.</p> <p>maxlength(optional) - Affects EditView only. Override to set the display maxlength attribute of the input field (defaults to 255).</p>

Float

The Float field is responsible for rendering a field that is formatted as a floating point number. The precision specified will be followed, or the user's default precision will take over.

Within the metadata definition, the Float field can be rendered with the snippet:

```
'weight', // Assumes that weight is defined
// as a float field in vardefs.php file
or
array('name'=>'weight',
'displayParams'=>array('precision'=>1)
),
```

name	Standard name definition when metadata definition is defined as an array
displayParams	<p>required (optional) - Marks the field as required and applies clients side validation to ensure value is present when save operation is invoked from EditView (Overrides the value set in vardefs.php).</p>



precision (optional) – Sets the displayed precision of the field, this specifies the number of digits after the decimal place that the field will attempt to format and display. Some databases may further limit the precision beyond what can be specified here.

Html

The Html field is a simple field that renders read-only content after the content is run through the `from_html` method of `include/utils/db_utils.php` to encode entity references to their HTML characters (i.e. ">" => ">"). The rendering of the Html field type should be handled by the custom field logic within SugarCRM.

name	Standard name definition when metadata definition is defined as an array.
displayParams	<i>none</i>

Iframe

In the DetailView, the Iframe field creates an HTML `<iframe>` element where the `src` attribute points to the given URL value supplied in the EditView. Alternatively, the URL may be generated from the values of other fields, and the base URL in the `vardefs`. If this is the case, the field is not editable in the EditView.

name	Standard name definition when metadata definition is defined as an array.
displayParams	<i>None</i>

Image

Similar to the Html field, the Image field simply renders a `` tag where the `src` attribute points to the value of the field.

Within the metadata file, the Image field can be rendered with the snippet:

```
array (
  'name' => 'my_image_value', // <-- The value of this is assumed to be some
  // URL to an image file
  'type' => 'image',
  'displayParams'=>array('width'=>100, 'length'=>100,
  'link'=>'http://www.cnn.com', 'border'=>0),
),
```

name	Standard name definition when metadata definition is defined as an array
displayParams	link (optional) - Hyperlink for the image when clicked on.



width (optional) - Width of image for display. height (optional) - Height of image for display. border (optional) - Border thickness of image for display

Int

The Int field is responsible for rendering a field that is formatted as a decimal point number. This will always be displayed as a whole number, and any digits after the decimal point will be truncated.

Within the metadata definition, the Int field can be rendered with the snippet:

```
'quantity', // Assumes that quantity is defined
// as a int field in vardefs.php file
or
array('name'=>'quantity',
'displayParams'=>array('required'=>1)
),
```

Link

The link field simply generates an <a> HTML tag with a hyperlink to the value of the field for DetailViews. For EditViews, it provides the convenience of pre-filling the "http://" value. Alternatively, the URL may be generated from the values of other fields, and the base URL in the vardefs. If this is the case, the field is not editable in EditView.

Within the metadata file, the Link field can be rendered with the snippet:

```
array (
'name' => 'my_image_value', // <-- The value of this is assumed to
// be some URL to an image file
'type' => 'link',
'displayParams'=>array('title'=>'LBL_MY_TITLE'),
),
```

name	Standard name definition when metadata definition is defined as an array
displayParams	required (optional) - Marks the field as required and applies clients side validation to ensure value is present when save operation is invoked from EditView (overrides the value set in vardefs.php). title (optional for detailviewdefs.php only) - The <a> tag's title attribute for browsers to display the link in their status area size (optional) - Affects EditView only. Override to set the display length attribute of the input field. maxlength(optional) - Affects EditView only. Override to set the display maxlength attribute of the input field (defaults to 255).



Multienum

The Multienum fields renders a bullet list of values for DetailViews, and renders a <select> form element for EditViews that allows multiple values to be chosen. Typically, the custom field handling in Sugar will map multienum types created through Studio, so you do not need to declare metadata code to specify the type override. Nevertheless, within the metadata file, the Multienum field can be rendered with the snippet:

```
array (  
'name' => 'my_multienum_field',  
'type' => 'multienum',  
'displayParams' => array('javascript'=>'onchange="alert(\'hello world!\');')  
)
```

name	Standard name definition when metadata definition is defined as an array
displayParams	size (optional) - Controls the size of the field (affects SearchForm control on browser only). Defaults to value of 6 for SearchForm. required (optional) - Marks the field as required and applies clients side validation to ensure value is present when save operation is invoked from EditView (Overrides the value set in vardefs.php). javascript (optional) - Custom JavaScript to embed within the <select> tag element (see above example for onchange event implementation).

Parent

The parent field combines a blend of a dropdown for the parent module type, and a text field with code to allow Quicksearch for quicksearch-enabled modules (see *include/SugarFields/Fields/Parent/EditView.tpl* file contents and JavaScript code for more information on enabling Quicksearch). There are also buttons to invoke popups and a button to clear the value. Because the parent field assumes proper relationships within the SugarCRM modules, it is not a field you can add through Studio or attempt to type override in the metadata files.

Password

The password field simply renders an input text field with the type attribute set to "password" to hide user input values. It is available to EditViews only.

name	Standard name definition when metadata definition is defined as an array
displayParams	required (optional) - Marks the field as required and applies clients side validation to ensure value is present when Save operation is invoked from EditView (Overrides the value set in vardefs.php). size (optional) - Override to set the display length attribute of the input field (defaults to 30).



Phone

The phone field simply invokes the callto:// URL references that could trigger Skype or other VOIP applications installed on the user's system. It is rendered for DetailViews only.

Radioenum

The Radioenum field renders a group of radio buttons. Radioenum fields are similar to the enum field, but only one value can be selected from the group.

Readonly

The readonly field simply directs EditView calls to use the SugarField's DetailView display. There are no Smarty .tpl files associated with this field.

Relate

The Relate field combines a blend of a text field with code to allow Quick Search. The quicksearch code is generated for EditViews (see include/TemplateHandler/TemplateHandler.php). There are also buttons to invoke popups and a button to clear the value. For DetailViews, the Relate field creates a hyperlink that will bring up a DetailView request for the field's value.

Within the metadata file, the Relate field can be rendered with the snippet:

```
array (  
  array('name'=>'account_name',  
    'type'=>'relate',  
    'displayParams'=>array('allowNewValue'=>true)  
  ),  
),
```

This will create a relate field that allows the user to input a value not found in the quicksearch list.

name	Standard name definition when metadata definition is defined as an array
displayParams	<p>required (optional) - Marks the field as required and applies clients side validation to ensure value is present when Save operation is invoked from EditView (Overrides the value set in vardefs.php).</p> <p>readOnly (optional for editviewdefs.php file only) - Makes the text field input area readonly so that you have to just use the popup selection.</p> <p>popupData - This field is generated for you by default. See include/SugarFields/Fields/SugarFieldRelate.php for more information. You should not need to override this setting.</p> <p>allowNewValue (optional for editviewdefs.php file only) - This setting allows the user to specify a value that is not found from the quicksearch list of results.</p> <p>hideButtons (optional for editviewdefs.php SearchForm.php and popupdefs.php) - Hides the Select and Clear buttons normally displayed next to the editable Relate field.</p>



Text

The Text field renders a <textarea> HTML form element for EditViews and displays the field value with newline characters converted to HTML elements in DetailViews.

Name	Standard name definition when metadata definition is defined as an array
displayParams	required (optional) - Marks the field as required and applies clients side validation to ensure value is present when Save operation is invoked from EditView (Overrides the value set in vardefs.php). maxlength (optional for editviewdefs.php file only) - Sets the maximum length of character input allowed in the <textarea> field. rows (optional for editviewdefs.php file only) - Sets the number of rows in the <textarea>field. cols (optional for editviewdefs.php file only) - Sets the number of cols in the <textarea> field.

Username

The Username field is a helper field that assumes a *salutation*, *first_name* and *last_name* field exists for the vardefs of the module. It displays the three fields in the format:
[salutation] [first_name] [last_name]

Metadata Framework Summary

In summary, the new metadata framework simplifies the management of the detail and EditViews by reducing the number of individual .tpl or .html files currently used in 4.X versions and prior. The problem was that a change to a module's view required the editing of the module's .html or .tpl file and with that, the extra checks against malformed html or smarty tags as well as proper field type displays (select elements for enum fields, checkboxes for booleans, etc.). By moving to a metadata driven framework, the fields that are rendered are tied directly to the module's vardefs.php file definition (barring any customization).

Formatting Changes

All numeric fields will be stored in the bean in an unformatted way. Any time a field is displayed to the user, formatting will have to be done. Each of the Sugar fields has a formatField function that you can call for this specific purpose.

Vardefs

Vardefs (Variable Definitions) are used to provide the Sugar application with information about SugarBeans. They specify information on the individual fields, relationships between beans, and the



indexes for a given bean. Each module that contains a SugarBean file has a vardefs.php file located in it, which specifies the fields for that SugarBean. For example, the Vardefs for the Contact bean is located in `sugarcrm/modules/Contacts/vardefs.php`.

Vardef files create an array called "\$dictionary", which contains several entries including tables, fields, indices, and relationships.

Dictionary Array

- 'table' = The name of the database table (usually, the name of the module) for this bean that contains the fields.
- 'audited' = Set to *True* if this module has auditing turned on
- 'fields' = A list of fields and their attributes (see below)
- 'indices' = A list of indexes that should be created in the database (see below)
- 'relationships' = A list of the relationships for this bean (see below)
- 'optimistic_locking' = True if this module should obey optimistic locking. Optimistic locking uses the modified date to ensure that the bean you are working on has not been modified by anybody else when you try and save. This prevents loss of data.
- 'unified_search' = True if this module is available to be searched thru the Global Search, defaults to false. Has no effect if none of the fields in the Fields array have the 'unified_search' attribute set to true.
- 'unified_search_default_enabled' – True if this module should be searched by default for new users thru the Global Search, defaults to true. Has no effect if 'unified_search' is not set to true.

Fields Array

The fields array contains one array for each field in the SugarBean. At the top level of this array the key is the name of the field, and the value is an array of attributes about that field.

The list of possible attributes are as follows:

- 'name' = The name of the field
- 'vname' = The language pack id for the label of this field
- 'type' = The type of the attribute



- o 'relate' = Related Bean
- o 'datetime' = A date and time
- o 'bool' = A boolean value
- o 'enum' = An enumeration (drop down list from the language pack)
- o 'char' = A character array
- o 'assigned_user_name' = A linked user name
- 'varchar' = A variable sized string
- 'table' = The table this field comes from
- 'isnull' = Is this field allowed to be set to null?
- 'len' = The length of the field (number of characters if a string)
- 'options' = The name of the enumeration in the language pack for this field
- 'dbtype' = The database type of the field (if different than the type)
- 'reportable' = Should this field show up in the list of fields for the reporting module (if applicable).
- 'required' = true if this field is a required field. If the field is placed on an edit view, the field must be populated before the user can save.
- 'default' = The default value for this field
- 'massupdate' = false if you do not want this field to show up in the mass update section at the bottom of the list views. Defaults to true.
- 'rname' = (for type relate only) The field from the related variable that has the text
- 'id_name' = (for type relate only) The field from the bean that stores the id for the related Bean



- 'source' = 'nondb' if the field value does not come from the database. This can be used for calculated values or values retrieved in some other way.
- 'sort_on' => The field to sort by if multiple fields are used.
- 'fields' => (for concatenated values only) An array containing the fields that are concatenated.
- 'db_concat_fields'=> (for concatenated values only) An array containing the fields to concatenate in the DB.
- 'unified_search' => True if this field should be searched through Global Search, defaults to false. Has no effect if the Dictionary array setting 'unified_search' is not set to true.

The following example illustrates a standard ID field for a Bean.

```
'id' => array (
  'name' => 'id',
  'vname' => 'LBL_ID',
  'type' => 'id',
  'required'=>true,
),
```

Indices Array

This array contains a list of arrays that are used to create indexes in the database. The fields in this array are:

- o 'name' = The name of the index. This must be unique in most databases.
- o 'type' = The type of the index (primary, unique, or index)
- o 'fields' = The fields to index. This is an ordered array.

The following example is to create a primary index called 'userspk' on the 'id' column

```
array('name' =>'userspk', 'type' =>'primary', 'fields'=>array('id')),
```

Relationships Array

The relationships array is used to specify relationships between Beans. Like the Indices array entries, it is a list of names with array values.

- 'lhs_module' = The module on the left hand side of the relationship
- 'lhs_table' = The table on the left hand side of the relationship



- 'lhs_key' = The primary key column of the left hand side of the relationship
- 'rhs_module' = The module on the right hand side of the relationship
- 'rhs_table' = The table on the right hand side of the relationship
- 'rhs_key' = The primary key column of the right hand side of the relationship
- 'relationship_type' = The type of relationship ('one-to-many' or 'many-to-many')
- 'relationship_role_column' = The type of relationship role
- 'relationship_role_column_value' = Defines the unique identifier for the relationship role

The following example creates a reporting relationship between a contact and the person that they report to. The reports_to_id field is used to map to the id field in the contact of the person they report to. This is a one-to-many relationship in that each person is only allowed to report to one person. Each person is allowed to have an unlimited number of direct reports.

```
'contact_direct_reports' => array(
  'lhs_module' => 'Contacts',
  'lhs_table' => 'contacts',
  'lhs_key' => 'id',
  'rhs_module' => 'Contacts',
  'rhs_table' => 'contacts',
  'rhs_key' => 'reports_to_id',
  'relationship_type' => 'one-to-many'),
```

Many-to-Many Relationships

In the *./metadata* directory, all the many-to-many relationships are defined and included in the \$dictionary array. The files are stored in *./metadata/<relation_table_name>MetaData.php*. Tables are generated based on these definitions. These files are included through the *./modules/TableDictionary.php*. If you create a custom many-to-many relationship, you will need to add the reference to the new relationship by adding the new reference in the file *custom/application/Ext/TableDictionary/tabledictionary.php*. You may need to create this file if it does not exist. These changes will take effect after you clear the Sugar Cache by running the "Quick Repair and Rebuild" option from the Admin Repair screen.

The following are the definitions for *\$dictionary[<relation_table>]*. They are similar to the Vardefs. If necessary use that page as a reference as well.

- **<relation_table>** - the index for this relationship in the \$dictionary array
- **table** - the name of the table that is created in the database



- **fields** - array containing arrays for each column definition. The join table must have a field for the primary key of each table to be linked, a primary key for the join table itself, a deleted field for relationship unlinking, and a date_modified field to track when the relationship changes. Additional fields can be added to track the role played by the relationship,
- **indices** - the database indices. Note see the example for indices below for necessary values.
- **relationships** - definitions of the relationships between the two tables
 - o **lhs_module** - the left hand module. Should match *\$beanList* index
 - o **lhs_table** - the left hand table name
 - o **lhs_key** - the key to use from the left table
 - o **rhs_module** - the right hand module. Should match *\$beanList* index
 - o **rhs_table** - the right hand table name
 - o **rhs_key** - the key to use from the right table
 - o **relationship_type** - relationship type
 - o **join_table** - join table used to join items
 - o **join_key_lhs** - left table key. Should exist in table field definitions above
 - o **join_key_rhs** - right table key. Should exist in table field definitions above

Note that Relationship metadata and the Vardefs are the critical building blocks of the new ListViews. In conjunction with [module]/metadata/ListViewdefs.php, these three elements generate your ListView.

For example, you may need to display data from another module or even just another table (not part of a module). To do this, you will need to get all your queries in order () AND add the reference in Vardefs.php. The new ListViews can *only* display data registered in the Vardefs.

For example:

```
'store_name' => array ( 'name' => 'store_name',
  'rname' => 'name',
  'id_name' => 'id',
  'vname' => 'LBL_STORE_NAME',
```



```

        'type' => 'relate',
        'link'=>'store_name',
        'table' => 'jd_stores',
        'join_name'=>'stores',
        'isnull' => 'true',
        'module' => 'Stores',
        'dbType' => 'varchar',
        'len' => 100,
        'source'=>'non-db',
        'unified_search' => false,
        'massupdate' => false,
        'comment' => 'The name of the store represented by the store_id field in customer_service'
    ),

```

In this case, Stores is not a module, but jd_stores is a table. The joins with that table are handled in fill_in_additional_detail_fields(), fill_in_additional_list_fields(), create_list_query(), and (depending on your situation) create_new_list_query(). You need to get all these things straight, but you will still not get your data on your ListView if you do not register it in Vardefs.

Subpanels

Subpanels are used in a given module's DetailView to display relationships with other modules. Examples include the Contacts subpanel under the Opportunities module's DetailView, or the Tasks subpanel for the Projects module.

The references below are the areas in the code where subpanels are generated from. Running rebuild relationships in the repair section in the admin panel is necessary after making changes to or creating subpanels.

The module that contains the subpanel is where the vardefs array index is defined. There is an index referring to the module that will appear as the subpanel of type 'link'.

One-to-Many Relationships

For Accounts, the reference necessary for the Cases subpanel is defined as follows in the *./modules/Accounts/vardefs.php*

```

    'cases' => array (
        'name' => 'cases',
        'type' => 'link',
        'relationship' => 'account_cases', //relationship definition is below
        'module'=>'Cases',
        'bean_name'=>'aCase',
        'source'=>'non-db',
        'vname'=>'LBL_CASES',
    ),

```

For a one-to-many, the 'relationship' index defined above must also be in the vardefs.

```

    'account_cases' => array(

```



```

'lhs_module'=> 'Accounts',
'lhs_table'=> 'accounts',
'lhs_key' => 'id',
'rhs_module'=> 'Cases',
'rhs_table'=> 'cases',
'rhs_key' => 'account_id',
'relationship_type'=>'one-to-many'
),

```

Since this is a one-to-many, there is no need for a relationship table, which is only defined in the *./metadata* directory.

Many-to-Many Relationships

For Accounts, the reference necessary for the Bugs subpanel is defined as follows in the *./modules/Accounts/vardefs.php*

```

'bugs' => array (
'name' => 'bugs',
'type' => 'link',
'relationship' => 'accounts_bugs', //relationship table
'module'=>'Bugs',
'bean_name'=>'Bug',
'source'=>'non-db',
'vname'=>'LBL_BUGS',
),

```

Since this is many-to-many relationship, and there already exists a relationship table, there is no need to define the relationship in the vardefs. However, the relationship metadata must be defined as shown below.

Relationship Metadata

If you have a many-to-many relationship, a table must exist for the relationship. For a new relationship, you must add the details of the relationship file (*accounts_bugsMetaData.php* in this example) to *TableDictionary.php* in the */modules* directory. You must then run Repair Database to create the new relationships table (*accounts_bugs* in the example below). To remain upgrade-safe you must put your custom changes into */custom/application/ext/tabledictionary/tabledictionary.ext.php*.

In the *./metadata* directory, the relationship must exist and included in the *\$dictionary* array. To keep consistent with the above *accounts_bugs* example, here is the content of the

accounts_bugsMetaData.php

```

$dictionary['accounts_bugs'] = array(
'table' => 'accounts_bugs', //the table that is created in the database
'fields' => array (
array('name' =>'id', 'type' =>'varchar', 'len'=>'36'), // id for relationship
array('name' =>'account_id', 'type' =>'varchar', 'len'=>'36'), // account id
array('name' =>'bug_id', 'type' =>'varchar', 'len'=>'36'), // bug id
array('name' => 'date_modified','type' => 'datetime'), // necessary
array('name' =>'deleted', 'type' =>'bool', 'len'=>'1', 'required'=>true, 'default'=>'0') // necessary
), // the indices are necessary for indexing and performance
'indices' => array (

```



```

array('name' => 'accounts_bugspk', 'type' => 'primary', 'fields' => array('id')),
array('name' => 'idx_acc_bug_acc', 'type' => 'index', 'fields' => array('account_id')),
array('name' => 'idx_acc_bug_bug', 'type' => 'index', 'fields' => array('bug_id')),
array('name' => 'idx_account_bug', 'type' => 'alternate_key', 'fields' => array('account_id', 'bug_id'))
),
'relationships' => array(
'accounts_bugs' => array(
'lhs_module' => 'Accounts', // the left hand module - should match $beanList index
'lhs_table' => 'accounts', // the left hand table name
'lhs_key' => 'id', // the key to use from the left table
'rhs_module' => 'Bugs', // the right hand module - should match $beanList index
'rhs_table' => 'bugs', // the right hand table name
'rhs_key' => 'id', // the key to use from the right table
'relationship_type' => 'many-to-many', // relationship type
'join_table' => 'accounts_bugs', // join table - table used to join items
'join_key_lhs' => 'account_id', // left table key - should exist in table field definitions above
'join_key_rhs' => 'bug_id' // right table key - should exist in table field definitions above
)
)
)
)

```

Layout Defs

This is the file that contains the related modules to create subpanels for. It is stored in the `$layout_defs` array `$layout_defs[<module>][subpanel_setup][<related_module>]`.

This example is from the `account metadata/subpaneldefs.php`

```

'contacts' => array(
'order' => 30, // the order in which this subpanel is displayed with other subpanels
'module' => 'Contacts',
'subpanel_name' => 'default', // in this case, it will use ./modules/Contacts/subpanels/default.php
'get_subpanel_data' => 'contacts',
'add_subpanel_data' => 'contact_id',
'title_key' => 'LBL_CONTACTS_SUBPANEL_TITLE',
'top_buttons' => array( // this array defines the top buttons
array('widget_class' => 'SubPanelTopCreateAccountNameButton'),
array('widget_class' => 'SubPanelTopSelectButton', 'mode' => 'MultiSelect')
),
),
)

```

In the language file for the module containing the subpanel, the following values need to be added.

- The reference used in 'title_key' as shown above
- In the example, it would be `$mod_strings['LBL_CONTACTS_SUBPANEL_TITLE'] = 'Contacts';`
- The reference used in 'vname' as shown in the vardefs section

In the example, it would be `$mod_strings['LBL_CASES'] = 'Cases';`



Shortcuts

Menu shortcuts for modules are easy to implement.. There is a `./modules/MODULE_NAME/Menu.php` in all applicable modules. Shortcuts are generated through:

```
$module_menu[]=Array("URLLINK", "SHORTCUTTEXT", "IMAGEFILENAME");
```

- URLLINK - the link that the shortcut points to
- SHORTCUTTEXT - the text that displays in the menu
- IMAGEFILENAME - the filename in the themes image directory
Note: A .gif extension is required for the image file and assumed in the definition.

Example from `./modules/Cases/Menu.php`

```
$module_menu[] =  
Array("index.php?module=Cases&action=index&return_module=Cases&return_action=DetailView",  
$mod_strings['LNK_CASE_LIST'], "Cases")
```

Copyright 2004-2010 SugarCRM Inc.

[Community Edition License](#)



Customizing Sugar

1. [Overview](#)
2. [Introduction](#)
3. [Tips](#)
 - 3.1. [Making upgrade-safe customizations](#)
 - 3.2. [Installing Third-Party Modules](#)
 - 3.3. [Naming Your Custom Modules](#)
 - 3.4. [Be Familiar with Object Oriented Programming](#)
 - 3.5. [Use Developer Mode when Customizing the User Interface](#)
4. [The Custom Directory](#)
 - 4.1. [Vardefs](#)
 - 4.1.1. [Master Directories](#)
 - 4.1.2. [Production Directories](#)
 - 4.1.3. [Description](#)
 - 4.2. [Languages](#)
 - 4.2.1. [Master Directories](#)
 - 4.2.2. [Production Directories](#)
 - 4.2.3. [Description](#)
 - 4.3. [Shortcuts](#)
 - 4.3.1. [Master Directories](#)
 - 4.3.2. [Production Directories](#)
 - 4.3.3. [Description](#)
 - 4.4. [Layoutdefs](#)
 - 4.4.1. [Master Directories](#)
 - 4.4.2. [Production Directories](#)
 - 4.4.3. [Rule](#)
 - 4.4.4. [Description](#)
5. [Module Builder](#)
 - 5.1. [Creating New Modules](#)
 - 5.2. [Understanding Object Templates](#)
 - 5.3. [Editing Module Fields](#)
 - 5.4. [Editing Module Layouts](#)
 - 5.5. [Building Relationships](#)
 - 5.6. [Publishing and Uploading Packages](#)
 - 5.7. [Adding Custom Logic using Code](#)



- 6. [Logic Hooks](#)
 - 6.1. [Custom Bean files](#)
 - 6.2. [Module Loader](#)
 - 6.2.1. [Manifest Overview](#)
 - 6.2.2. [Installdef Definition](#)
 - 6.2.3. [Upgrade Definition](#)
- 7. [Module Loader Restrictions](#)
 - 7.1. [Access Controls](#)
 - 7.2. [Enable Package Scan](#)
 - 7.3. [Enable File Scan](#)
 - 7.4. [Disable Module Loader Actions](#)
 - 7.4.1. [Restricted Copy](#)
- 8. [Business Logic Hooks](#)
 - 8.1. [Hook Definition](#)
 - 8.2. [Module hooks](#)
 - 8.3. [Hooks for Users module](#)
 - 8.4. [Options Array](#)
 - 8.5. [Packaging Custom Logic Hooks](#)
 - 8.5.1. [Custom Grouping of Values](#)
 - 8.5.2. [Creating New Custom Displays](#)
 - 8.6. [Overriding the View](#)
- 9. [Creating a Custom Sugar Field](#)
 - 9.1. [Adding QuickSearch to a Custom Field](#)
 - 9.2. [Tips](#)
 - 9.2.1. [Grouping Required Fields Together](#)
- 10. [Creating New Sugar Dashlets](#)
 - 10.1. [Custom Sugar Dashlets](#)
 - 10.2. [Packaging Custom Sugar Dashlets](#)
- 11. [Themes](#)
 - 11.1. [Theme Directory Structure](#)
 - 11.2. [Theme Development](#)
 - 11.3. [Creating a New Theme](#)
 - 11.4. [Element Reference Guide](#)
 - 11.5. [Packaging Custom Themes](#)
 - 11.6. [Tips](#)
 - 11.6.1. [Pick your Canvas](#)
 - 11.6.2. [Replace All](#)
 - 11.6.3. [Check your work](#)



- 11.6.4. [Personalize your theme](#)
- 11.6.5. [Adding Multiple Languages](#)
- 11.6.6. [Add a Language](#)
- 12. [Creating Language Packs](#)
- 13. [Dynamic Teams](#)
 - 13.1. [Team Security](#)
 - 13.2. [Team Sets](#)
 - 13.3. [Primary Team](#)
 - 13.3.1. [Removing Teams Programmatically](#)
 - 13.3.2. [Replacing Teams Programmatically](#)
- 14. [Printing to PDF](#)
 - 14.1. [SugarPDF Architecture](#)
 - 14.2. [Key Classes](#)
 - 14.2.1. [ViewSugarpdf \(include/MVC/View/views/view.sugarpdf.php\)](#)
 - 14.2.2. [TCPDF \(include/tcpdf/tcpdf.php\)](#)
 - 14.2.3. [Sugarpdf \(include/Sugarpdf/Sugarpdf.php\)](#)
 - 14.2.4. [Sugarpdf.XXX](#)
 - 14.2.5. [SugarpdfFactory](#)
 - 14.2.6. [SugarpdfHelper](#)
 - 14.2.7. [FontManager](#)
 - 14.3. [PDF settings \(user and system\)](#)
 - 14.4. [The custom directory](#)
 - 14.5. [Adding New PDF Templates](#)
 - 14.6. [Smarty](#)
- 15. [How to Create a Portal API User](#)
- 16. [How to Enable Unsupported Email Configurations](#)

Overview

[You can customize Sugar to tailor the application to meet your business needs. This chapter explains the different ways to customize SugarCRM.](#)

Introduction

[The extension framework in Sugar was created to help implement the customization of existing modules or create entirely new modules. Through the various extensions available you can extend most of the functionality of Sugar in an upgrade-safe manner. The Module Builder and Studio tools, available from](#)



[the Admin Home page](#), allow you to make the most common customizations that are outlined below. You can then further extend your system by [adding upgrade-safe custom code](#).

[Most common customizations are done with the suite of developer tools provided in the Sugar Admin screen](#). Those tools include:

- [Studio](#) - Edit Dropdowns, Custom Fields, Layouts and Labels
- [Module Builder](#) - Build new modules to expand the functionality of SugarCRM
- [Module Loader](#) - Add or remove Sugar modules, themes, and language packs
- [Dropdown Editor](#) - Add, delete, or change the dropdown lists in the application
- [Rename Tabs](#) - Change the label of the module tabs
- [Configure Module Tabs and Subpanels](#) - Choose which module tabs and sub-panels to display within the application
- [Configure Shortcut Bar](#) - Select which modules are available in the Shortcuts bar.
- [Configure Grouped Modules](#) - Create and edit groupings of tabs

[For further information on how to use these tools, please refer to the Sugar Application Guide](#). This guide will go into more detail on how to use the Module Builder and the Module Loader, as well as how to extend the Sugar system at the code-level beyond what these tools provide.

[Tips](#)

[Making upgrade-safe customizations](#)

[Because Sugar is an open source application, you have access to the source code. Keep in mind that any code customizations you make to the core files that ship with the Sugar application will need to be merged forward or re-factored manually when you upgrade to the next patch or major release. However, any changes you make using the developer tools provided on the Admin screen \(Module Builder, Studio, etc.\) are upgrade-safe. Also, any code changes you make in the *custom/* directory are also upgrade-safe.](#)

[Installing Third-Party Modules](#)

[Be aware that not all third-party modules you install on your Sugar system \(such as modules found on SugarForge.org\) have been designed to interoperate error-free with other modules and, hence, may not be upgrade-safe. Code changes made by a third-party developer and distributed in a module could potentially modify core files which would require special attention during an upgrade. Also, two different modules could conflict with one another in the changes they make to your system.](#)



[Naming Your Custom Modules](#)

[If you create a new module or a Sugar Dashlet without using the Module Builder, be sure to give your new directories unique names. This will prevent conflicts with future modules added by the Sugar team. For example, a best practice would be to add a unique prefix to your new module's directory name such as "zz_Brokers" for a new Brokers module.](#)

[Be Familiar with Object Oriented Programming](#)

[Much of extending the Sugar functionality is based around extending and overriding the SugarCRM base classes. Developers need to be familiar with the basics of object-oriented programming and are encouraged to extend the SugarCRM base classes only to the minimum extent necessary.](#)

[Use Developer Mode when Customizing the User Interface](#)

[When developing in Sugar, we suggest that you turn on the **Developer Mode** \(Admin->System Settings->Advanced->Developer Mode\) to get the system to ignore the cached metadata files. This is especially helpful when you are directly altering templates, metadata, or language files. When using Module Builder or Studio, the system will automatically refresh the file cache. Be sure to turn off Developer Mode when you have completed your customizations since this mode does degrade system performance.](#)

[The Custom Directory](#)

[The Sugar system contains a top level directory called "custom" directory. This directory contains metadata files and custom code that override and extend the base Sugar functionality. Some of the files in this directory are auto-generated by the Module Builder, Studio, and Workflow tools \(Sugar Professional and Sugar Enterprise only\) and other files can be added or modified directly by a developer. Before discussing how a developer can use Sugar tools to modify the application, an understanding of the custom directory is useful.](#)

[Vardefs](#)

[Vardefs define field attributes for a given module. Existing vardefs can be modified and new vardefs can be created by modifying vardefs files in the custom directory.](#)

[Master Directories](#)

[Files in these directories can be edited and new files can be added to these directories.
/custom/Extension/modules/<MODULE_NAME>/Ext/Vardefs/](#)

[Production Directories](#)

[Files in these directories are auto-generated by the system and should not be modified.
/custom/modules/<MODULE_NAME>/Ext/Vardefs/vardefs.ext.php](#)

[Description](#)

[Vardefs files either replace field definitions entirely, or add to the ones that are available to a module. In the Master Directories you can have many files such as:](#)

- [/custom/Extension/modules/Calls/Ext/Vardefs/New_vardefs.php](#)



- /custom/Extension/modules/Calls/Ext/Vardefs/Updated_Vardefs.php

During the repair function (Admin->Repair->Quick Repair and Rebuild), all of these files will be merged together into the production directory and they become the file:

</custom/modules/Calls/Ext/Vardefs/vardefs.ext.php>

For example, a vardefs extension file for the Calls module could contain the following:

/custom/Extension/modules/Calls/ext/Vardefs/Import_Vardefs.php

```
<?php
dictionary['Call']['fields']['parent_type']['vname'] = 'LBL_PARENT_TYPE';
dictionary['Call']['fields']['parent_id']['vname'] = 'LBL_PARENT_ID';
dictionary['Call']['fields']['deleted']['importable'] = false;
dictionary['Call']['fields']['related_id'] = array (
  'name' => 'related_id',
  'vname' => 'LBL_RELATED_ID',
  'type' => 'id',
  'required' => false,
  'reportable' => false,
  'audited' => true,
  'comment' => 'ID of a related record of this call',
);
?>
```

This would change the values for the "vname" of parent_type and parent_id, it would add an "importable" value to the deleted field and set it to false and then add a whole new field called related_id to the calls field list.

Languages

It is possible to override display string values for a given language and create entirely new strings used by new custom fields.

Master Directories

Files in these directories can be edited and new files can be added to these directories.

- </custom/include/language/> (for *\$app_strings* or *\$app_list_strings*)
- </custom/Extension/application/Ext/Include/>
- /custom/Extension/modules/<MODULE_NAME>/Ext/Language/ (for *\$mod_strings* only)

Production Directories

Files in these directories are auto-generated by the system and should not be modified.

- /custom/include/language/<LANGUAGE_TAG>.lang.ext.php
- /custom/modules/<MODULE_NAME>/Ext/Languages/<LANGUAGE_TAG>.lang.ext.php



Description

[Language files either replace entirely or add to the translated language items available to a module. In the Master Directories you can have many files like:](#)

- [/custom/Extension/modules/Leads/Ext/Language/en_us.Custom_strings.php](#)
- [/custom/Extension/modules/Leads/Ext/Language/en_us.Custom_Languages.php](#)

[During the repair function \(Admin->Repair->Quick Repair and Rebuild\), all of these files will be merged together into the production directory and they become the file:](#)

[/custom/modules/Leads/Ext/Language/en_us.lang.ext.php](#)

[For example, a language extension file for the Calls module could contain the following:](#)

[/custom/Extension/modules/Calls/ext/Languages/en_us.Import_Menu.php](#)

```
<?php
// adding Import field changes
$mod_strings['LNK_IMPORT_CALLS'] = 'Import Calls';
$mod_strings['LBL_MODIFIED_NAME'] = 'Modified By';
$mod_strings['LBL_PARENT_TYPE'] = 'Parent Type';
$mod_strings['LBL_PARENT_ID'] = 'Parent ID';
?>
```

[This would add four new display strings to the Calls module.](#)

Shortcuts

[It is possible to override or create new Shortcuts menu items.](#)

Master Directories

[Files in these directories can be edited and new files can be added to these directories.](#)

- [/custom/Extension/application/Ext/Menus/](#)
- [/custom/Extension/modules/<MODULE_NAME>/Ext/Menus/](#)

Production Directories

[Files in these directories are auto-generated by the system and must not be modified.](#)

- [/custom/application/Ext/Menus/menu.ext.php](#)
- [/custom/modules/<MODULE_NAME>/Ext/Menus/menu.ext.php](#)

Description

[Shortcut menu files either replace entirely, or add to the menu items available under the "Actions" list on the module tabs. In the Master Directories you can have many files like:](#)

- [/custom/Extension/modules/Calls/Ext/Menus/New_menu_items.php](#)



- /custom/Extension/modules/Calls/Ext/Menus/Custom_Menu.php
- /custom/Extension/modules/Calls/Ext/Menus/Menu_items.php

During the repair function (Admin->Repair->Quick Repair and Rebuild), all of these files will be merged together into the production directory and they become the file:

</custom/modules/Leads/ext/Menus/menu.ext.php>

For example, a menu extension file for the Calls module could contain the following:

```
<?php
if(ACLController::checkAccess('Calls', 'import', true)) {
    $module_menu[]=Array("index.php?module=Calls&action=MakeIndex",
        translate('LNK_INDEX_CALLS'), "Import"
    );
}
?>
```

This would add a menu item to the Calls module's Shortcuts menu. The \$module_menu array takes three elements.

- [The first is the URL that the menu item will run.](#)
- [The second is the text that will be shown on the menu, in this case we added some custom language text in a separate custom language file that we will go over later in this document.](#)
- [The last is the name of the icon associated with this menu option. This word will have ".gif" added to the end of it. If you want to use a png file here you must rename import.png to import.gif and load it into the themes/default/images directory. Even though it is named with the 'gif' extension it will still work.](#)

If you added a **\$module_menu=array();** to the top of this file, you would effectively clear out any of the standard menu items. You could then replace them all with new definitions.

If the custom file is in the </custom/Extension/application/Ext/Menus/> directory, then your menu changes will affect shortcut menus in every module.

[Layoutdefs](#)

Master Directories

Files in these directories can be edited and new files can be added to these directories.

- </custom/Extension/application/Ext/Layoutdefs/>
- /custom/Extension/modules/<MODULE_NAME>/Ext/Layoutdefs/

Production Directories

Files in these directories are auto-generated by the system and must not be modified.



- [/custom/application/Ext/Layoutdefs/layoutdefs.ext.php](#)
- [/custom/modules/<MODULE_NAME>/Ext/Layoutdefs/layoutdefs.ext.php](#)

Rule

Use the following rule to add a sub-panel to a module:

`For{$modulename}.php`

where `modulename` is the name of the module to which you are adding the sub-panel.

Description

[Layoutdefs](#) are a little more complex than the other customization types. Each customization is made across two files, the layout definition file and the actual layout file. In the Master Directories you can have many files like the following:

- [/custom/Extension/modules/Accounts/Ext/Layoutdefs/_overrideAccountContactsForAccounts.php](#)
- [/custom/Extension/modules/Accounts/Ext/Layoutdefs/_overrideAccountOpportunitiesForAccounts.php](#)

During the repair function (Admin->Repair->Quick Repair and Rebuild), all of these files are merged together into the production directory to become the file:

[/custom/modules/Accounts/Ext/Layoutdefs/layoutdefs.ext.php](#)

For example, a layoutdefs extension file for the Accounts module could contain the following:

[/custom/Extension/modules/Accounts/ext/Layoutdefs/_overrideAccountContactsForAccounts.php](#)

```
<?php
$layout_defs["Accounts"]["subpanel_setup"]["accounts_documents"] = array (
  'order' => 100,
  'module' => 'Documents',
  'subpanel_name' => 'default',
  'sort_order' => 'asc',
  'sort_by' => 'id',
  'title_key' => 'LBL_ACCOUNTS_DOCUMENTS_FROM_DOCUMENTS_TITLE',
  'get_subpanel_data' => 'accounts_documents',
);

$layout_defs['Accounts']['subpanel_setup']['contacts']['override_subpanel_name'] =
'AccountForAccounts';
?>
```

The first array is setting up a subpanel for a new relationship between Documents and Accounts. While there are many other files required to completely define this module relationship, (which we will go over in the Relationship section below), this file just creates the links to the new subpanel that would be located at:

[custom/modules/Documents/Ext/subpanels/default.php](#)

The second array simply points the system at a new subpanel definition file that will replace the subpanel that shows Contacts related to Accounts. Since you cannot merge subpanel definition files, they do not exist in the custom/Extension/ directory. The array in this file would tell the system to load the file:

[custom/modules/Contacts/metadata/subpanels/AccountForAccounts.php](#)



Module Builder

The Module Builder functionality allows programmers to create custom modules without writing code, and to create relationships between new and existing CRM modules. To illustrate how to use Module Builder, this article will show how to create and deploy a custom module. In this example, a custom module to track media inquiries will be created to track public relations requests within a CRM system. This use case is an often requested enhancement for CRM systems that applies across industries.

Creating New Modules

Module Builder functionality is managed within the 'Developer Tools' section of Sugar's administration console.

Upon selecting 'Module Builder', the user has the option of creating a "New Package". Packages are a collection of custom modules, objects, and fields that can be published within the application instance or shared across instances of Sugar. Once the user selects "New Package", the user names and describes the type of Custom Module to be created. A package key, usually based on the organization or name of the package creator is required to prevent conflicts between two packages with the same name from different authors. In this case, the package will be named "MediaTracking" to explain its purpose, and a key based on the author name will be used.

Once the new package is created and saved, the user is presented with a screen to create a Custom Module. Upon selecting the "New Module" icon, a screen appears showing six different object templates.

Understanding Object Templates

Five of the six object templates contain pre-built CRM functionality for key CRM use cases. These objects are: "basic", "company", "file", "issue", "person", and "sale". The "basic" template provides fields such as Name, Assigned to, Team, Date Created, and Description. As their title denotes, the rest of these templates contain fields and application logic to describe entities similar to "Accounts", "Documents", "Cases", "Contacts", and "Opportunities", respectively. Thus, to create a Custom Module to track a type of account, you would select the "Company" template. Similarly, to track human interactions, you would select "People".

For the media tracking use case, the user will use the object template "Issue" because inbound media requests have similarities to incoming support cases. In both examples, there is an inquiry, a recipient of the issue, assignment of the issue and resolution. The final object template is named "Basic" which is the default base object type. This allows the administrator to create their own custom fields to define the object.

Upon naming and selecting the Custom Module template named "Issue", the user can further customize the module by changing the fields and layout of the application, and creating relationships between this new module and existing standard or custom modules. This Edit functionality allows user to construct a module that meets the specific data requirements of the Custom Module.

Editing Module Fields

Fields can be edited and created using the field editor. Fields inherited from the custom module's templates can be relabeled while new fields are fully editable. New fields are added using the "Add Field" button. This will bring up a tab where you can select the type of field to add as well as any properties that field type requires.



[Editing Module Layouts](#)

The layout editor can be used to change the appearance of the screens within the new module, including the EditView, DetailView and ListView screens. When editing the Edit View or the Detail View, new panels and rows can be dragged from the toolbox on the left side to the layout area on the right. Fields can then be dragged between the layout area and the toolbox. Fields are removed from the layout by dragging them from the layout area to the recycling icon. Fields can be expanded or collapsed to take up one or two columns on the layout using the plus and minus icons. List, Search, Dashlet, and Subpanel views can be edited by dragging fields between hidden/visible/available columns.

[Building Relationships](#)

Once the fields and layout of the Custom Module have been defined, the user then defines relationships between this new module and existing CRM data by clicking "View Relationships". The "Add Relationship" button allows the user to associate the new module to an existing or new custom module in the same package. In the case of the Media Tracker, the user can associate the Custom Module with the existing, standard 'Contacts' module that is available in every Sugar installation using a many-to-many relationship. By creating this relationship, end-users will see the Contacts associated with each Media Inquiry. We will also add a relationship to the activities module so that a Media Inquiry can be related to calls, meetings, tasks, and emails.

[Publishing and Uploading Packages](#)

After the user has created the appropriate fields, layouts, and relationships for the custom modules, this new CRM functionality can be deployed. Click the "Deploy" button to deploy the package to the current instance. This is the recommended way to test your package while developing. If you wish to make further changes to your package or custom modules, you should make those changes in Module Builder, and click the Deploy button again. Clicking the Publish button generates a zip file with the Custom Module definitions. This is the mechanism for moving the package to a test environment and then ultimately to the production environment. The Export button will produce a module loadable zip file, similar to the Publish functionality, except that when the zip file is installed, it will load the custom package into Module Builder for further editing. This is a good method for storing the custom package in case you would like to make changes to it in the future on another Sugar instance. After the new package has been published, the administrator must commit the package to the Sugar system through the Module Loader. The administrator uploads the files and commits the new functionality to the live application instance.

[Adding Custom Logic using Code](#)

While the key benefit of the Module Builder is that the Administrator user is able to create entirely new modules without the need to write code, there are still some tasks that require writing PHP code. For instance, adding custom logic or making a call to an external system through a Web Service. This can be done in one of two methods.

[Logic Hooks](#)

One way is by writing PHP code that leverages the event handlers, or "logic hooks", available in Sugar. In order to accomplish this, the developer must create the custom code and then add it to the manifest file for the "Media Inquiry" package.

Here is some sample code for a simple example of using the logic hooks. This example adds a time stamp to the description field of the Media Inquiry every time the record is saved.



First, create the file *AddTimeStamp.php* with the following contents.

```
<?php
//prevents directly accessing this file from a web browser
if(!defined('sugarEntry') || !sugarEntry) die('Not A Valid Entry Point');

-
class AddTimeStamp {
function StampIt(& $focus, $event){
global $current_user;
$focus->description .= "\nSaved on ". date("Y-m-d g:i a"). " by ". $current_user->user_name;
}
}
?>
```

Next register this custom function by creating the file *logic_hooks.php* with the following contents.

```
<?php
// Do not store anything in this file that is not part of the array or the hook version. This file will
// be automatically rebuilt in the future.
$hook_version = 1;
$hook_array = Array();
// position, file, function
$hook_array['before_save'] = Array();
$hook_array['before_save'][] = Array(1, 'custom', 'custom/modules/Media/AddTimeStamp.php
','AddTimeStamp', 'StampIt');
?>
```

Now add these two files to the Media Inquiries zip file you just saved. Create a directory called *"SugarModules/custom/"* in the zip file and add the two files there. Then modify the *manifest.php* in the zip file to include the following definition in the *\$install_defs['copy']* array.

```
array (
'from' => '<basepath>/SugarModules/custom',
'to' => 'custom/modules/jsche_Media',
),
```

Custom Bean files

Another method is to add code directly to the custom bean. This is a more complicated approach because it requires understanding the SugarBean class. However it is a far more flexible and powerful approach.

First you must "build" your module. This can be done by either deploying your module or clicking the Publish button. Module Builder will then generate a folder for your package in "custom/modulebuilder/builds/". Inside that folder is where Sugar will have placed the bean files for your new module(s). In this case we want

"custom/modulebuilder/builds/MediaTracking/SugarModules/modules/jsche_mediarequest"

Inside you will find two files of interest. The first one is *{module_name}sugar.php*. This file is generated by Module Builder and may be erased by further changes in module builder or upgrades to the Sugar application. You should not make any changes to this file. The second is *{module_name}.php*. This is the file where you make any changes you would like to the logic of your module. To add our timestamp, we would add the following code to *jsche_mediarequest.php*

```
function save($check_notify = FALSE) {
global $current_user;
$this->description .= "\nSaved on " . date("Y-m-d g:i a"). " by "
. $current_user->user_name;
```



```
parent::save($check_notify);  
}
```

The call to the `parent::save` function is critical as this will call on the out of box SugarBean to handle the regular Save functionality. To finish, re-deploy or re-publish your package from Module Builder. You can now upload this module, extended with custom code logic, into your Sugar application using the Module Loader as described earlier.

Using the New Module

After you upload the new module, the new custom module appears in the Sugar instance. In this example, the new module, named "Media" uses the object template "Issue" to track incoming media inquiries. This new module is associated with the standard "Contacts" modules to show which journalist has expressed interest. In this example, the journalist has requested a product briefing. On one page, users can see the nature of the inquiry, the journalist who requested the briefing, who the inquiry was assigned to, the status, and the description.

Module Loader

The Module Loader not only installs custom modules but installs the latest patches, themes, language packs, and Sugar Dashlets. The `$upgrade_manifest` variable is used to upgrade the application. The Module Loader relies on a file named `manifest.php`, which resides in the root directory of any installable package.

Manifest Overview

The following section outlines the parameters specified in the `$manifest` array contained in the manifest file (`manifest.php`). An example manifest file is included later for your reference.

`$manifest` array elements provide the Module Loader with descriptive information about the extension.

- o **acceptable_sugar_flavors** - Specifies which Sugar Editions the package can be installed on. Accepted values are any combination of: OS (valid prior to Sugar 5.0.0), CE (valid after Sugar 5.0.0), PRO, and ENT.
- o **acceptable_sugar_versions** - This directive contains two arrays:
 - o **exact_matches**: each element in this array should be one exact version string, i.e. "6.0.0b" or "6.1.0"
 - o **regex_matches**: each element in this array should be one regular expression designed to match a group of versions, i.e. "6\\.1\\.0[a-z]"
- o **author** - Contains the author of the package, i.e. "SugarCRM Inc."
- o **copy_files** - An array detailing the source and destination of files that should be copied during installation of the package. See the example manifest file below for details.
- o **dependencies** - A set of dependencies that must be satisfied to install the module. This contains two arrays:
 - o **id_name** : the unique name found in `$installdefs`.



- o [version: the version number.](#)

description - A description of the package. Displayed during installation.

- o Note: Apostrophes (') are not supported in your description and will cause a Module Loader error.

- o **icon** - A path (within the package ZIP file) to an icon image that will be displayed during installation. Examples include: `"/patch_directory/icon.gif"` and `"/patch_directory/images/theme.gif"`

- o **is_uninstallable** - Setting this directive to *TRUE* allows the Sugar administrator to uninstall the package. Setting this directive to *FALSE* disables the uninstall feature.

- o **name** - The name of the package. Displayed during installation. Note: Apostrophes (') are not supported in your description and will cause a Module Loader error.

- o **published_date** - The date the package was published. Displayed during installation.

- o **type** - The package type. Accepted values are:

- o [langpack - Packages of type langpack](#) will be automatically added to the "Language" dropdown on the Sugar Login screen. They are installed using the Upgrade Wizard.

- o [module - Packages of type module](#) are installed using the Module Loader.

- o [patch - Packages of type patch](#) are installed using the Upgrade Wizard.

- o [theme - Packages of type theme](#) will be automatically added to the "Theme" dropdown on the Sugar Login screen. They are installed using the Upgrade Wizard.

- o **version** - The version of the patch, i.e. "1.0" or "0.96-pre1" .

Installdef Definition

The following section outlines the parameters specified in the `$installdef` array contained in the manifest file (manifest.php). An example manifest file is included later for your reference.

`$installdef` array elements are used by the Module Loader to determine the actual installation steps that need to be taken to install the extension.

- **id** - A unique name for your module; for example, "Songs"



- **[copy](#)** - An array detailing files to be copied to the Sugar directory. A source path and destination path must be specified for each file or directory. See the example manifest file below for details.

- **[language](#)** - An array detailing individual language files for your module. The source path, destination file, and language pack name must be specified for each language file. See the example manifest file below for details.

- **[layoutdefs](#)** - An array detailing individual layoutdef files, which are used primarily for setting up subpanels in other modules. The source path and destination module must be specified for each layoutdef file. See the example manifest file below for details.

- **[layoutfields](#)** - An array detailing custom fields to be added to existing layouts. The fields will be added to the edit and detail views of target modules. See the example manifest file below for details.

- **[vardefs](#)** - An array detailing individual vardef files, which are used primarily for defining fields and non many-to-many relationships in other modules. The source path and destination module must be specified for each vardef file. See the example manifest file below for details.

- **[menu](#)** - An array detailing the menu file for your new module. A source path and destination module must be specified for each menu file. See the example manifest file below for details.

- **[beans](#)** - An array specifying the bean files for your new module. The following sub-directives must be specified for each bean:
 - o [module: Your module's name, "Songs"](#)
 - o [class: Your module's primary class name, "Song"](#)
 - o [path: The path to your bean file where the above class is defined.](#)
 - o [tab: Controls whether or not your new module appears as a tab.](#)

- **[relationships](#)** - An array detailing relationship files, used to link your new modules to existing modules. A metadata path must be specified for each relationship. See the example manifest file below for details.

- **[custom_fields](#)** - An array detailing custom fields to be installed for your new module. The following sub-directives must be specified for each custom field:



- ? [name: The internal name of your custom field. Note that your custom field will be referred to as <name>_c, as "_c" indicates a custom field.](#)
- ? [label: The visible label of your custom field](#)
- ? [type: The type of custom field. Accepted values include text, textarea, double, float, int, date, bool, enum, and relate.](#)
- ? [max_size: The custom field's maximum character storage size](#)
- ? [require_option: Used to mark custom fields are either required or option. Accepted values include optional and required.](#)
- ? [default_value: Used to specify a default value for your custom field](#)
- ? [ext1: Used to specify a dropdown name \(only applicable to enum type custom fields\)](#)
- ? [ext2: Unused](#)
- ? [ext3: Unused](#)
- ? [audited: Used to denote whether or not the custom field should be audited. Accepted values include 0 and 1.](#)
- ? [module: Used to specify the module where the custom field will be added.](#)

Upgrade Definition

[This array definition describes the release\(s\) leading to the "version" element defined in \\$manifest.](#) In the sample manifest below it defines the Songs module versions 1.0 and 1.5 as preceding the current version 2.0.

Module Loader Restrictions

[SugarCRM's hosting objective is to maintain the integrity of the standard Sugar functionality when we upgrade a customer instance, and limit any negative impact our upgrade has on the customer's modifications.](#)

Access Controls

[The Module Loader includes a Module Scanner, which grants system administrators the control they need to determine the precise set of actions that they are willing to offer in their hosting environment. This](#)



[feature is available in all editions of Sugar. Anyone who is hosting Sugar products can advantage of this feature as well.](#)

[The specific Module Loader restrictions for the Sugar Open Cloud are documented in the Sugar Knowledge Base.](#)

[Enable Package Scan](#)

[Scanning is disabled in default installations of Sugar, and can be enabled through a configuration setting. This setting is added to config.php or config_override.php, and is not available to Administrator users to modify through the Sugar interface.](#)

[To enable Package Scan and its associated scans, add this setting to config.php or config_override.php:](#)
`$GLOBALS['sugar_config']['moduleInstaller']['packageScan'] = true;`

[There are two categories of access controls now available:](#)

1. [File scanning](#)
 2. [Module Loader actions](#)
-

[Enable File Scan](#)

[By enabling Package Scan, File Scan will be performed on all files in the package uploaded through Module Loader. File Scan will be performed when a Sugar administrator attempts to install the package. File Scan performs two types of checks:](#)

1. [File extension must be in the approved list of valid extension types](#)
 - a. [The default list of valid extension types is detailed in Appendix A.](#)
 - b. [Files do not contain function calls that are considered suspicious, based on a blacklist.](#)
 - i. [Backticks \(` \) are never allowed by File Scan.](#)
 - ii. [The default blacklist of functions is detailed in Appendix B.](#)

[To disable File Scan, add the following configuration setting to config.php or config_override.php:](#)
`$GLOBALS['sugar_config']['moduleInstaller']['disableFileScan'] = true;`

[To add more file extensions to the approved list of valid extension types, add the file extensions to the validExt array. The example below adds a .log file extension and .htaccess to the valid extension type list:](#)

```
$GLOBALS['sugar_config']['moduleInstaller']['validExt'] = array('log', 'htaccess');
```

[To add additional function calls to the black list, add the function calls to the blacklist array. The example below blocks the strlen\(\) and strtolower\(\) functions from being included in the package:](#)

```
$GLOBALS['sugar_config']['moduleInstaller']['blackList'] = array('strlen', 'strtolower');
```



[To override the black list and allow a specific function to be included in packages, add the function call to the blackListExempt array. The example below removes the restriction for the file_put_contents\(\) function, allowing it to be included in the package:](#)

```
$GLOBALS['sugar_config']['moduleInstaller']['blackListExempt'] = array('file_put_contents');
```

Disable Module Loader Actions

[Certain Module Loader actions may be considered less desirable than others by a System Administrator. A System Administrator may want to allow some Module Loader actions, but disable specific actions that could impact the upgrade-safe integrity of the Sugar instance.](#)

[By default, all Module Loader actions are allowed. Enabling Package Scan does not affect the Module Loader actions.](#)

[To disable specific Module Loader actions, add the action to the disableActions array. The example below restricts the pre_execute and post_execute actions:](#)

```
$GLOBALS['sugar_config']['moduleInstaller']['disableActions'] = array('pre_execute',  
'post_execute');
```

[A list of all actions available in Module Loader is detailed in Appendix C.](#)

```
$GLOBALS['sugar_config']['disable_uw_upload'] = true;
```

[This configuration parameter blocks the upload capabilities of the Upgrade Wizard, intended for hosting providers. It behaves similarly to the use_common_ml_dir parameter for Module Loader.](#)

Restricted Copy

[To ensure upgrade-safe customizations, it is necessary for system administrators to restrict the copy action to prevent modifying the existing Sugar source code files. New files may be added anywhere \(to allow new modules to be added\), but any core Sugar source code file must not be overwritten. This is enabled by default when you enable Package Scan.](#)

[To disable Restricted Copy, use this configuration setting:](#)

```
$GLOBALS['sugar_config']['moduleInstaller']['disableRestrictedCopy'] = true;
```

Default Valid File Extensions

- [png](#)
- [gif](#)
- [jpg](#)
- [css](#)
- [js](#)
- [php](#)
- [txt](#)



- [html](#)
- [htm](#)
- [tpl](#)
- [md5](#)
- [pdf](#)

Default Blacklist of Functions

[eval](#)
[exec](#)
[system](#)
[shell_exec](#)
[passthru](#)
[chgrp](#)
[chmod](#)
[chown](#)
[file_put_contents](#)
[file](#)
[fileatime](#)
[filectime](#)
[filegroup](#)
[fileinode](#)
[filemtime](#)
[fileowner](#)
[fileperms](#)
[fopen](#)
[is_executable](#)
[is_writable](#)
[is_writable](#)
[lchgrp](#)
[lchown](#)
[linkinfo](#)
[lstat](#)
[mkdir](#)
[parse_ini_file](#)
[rmdir](#)
[stat](#)
[tempnam](#)
[touch](#)
[ulink](#)
[getimagesize](#)
[copy](#)
[link](#)
[rename](#)
[symlink](#)
[move_uploaded_file](#)
[chdir](#)



[chroot](#)
[sugar_chown](#)
[sugar_fopen](#)
[sugar_mkdir](#)
[sugar_file_put_contents](#)
[sugar_chgrp](#)
[sugar_chmod](#)
[sugar_touch](#)

Module Loader Actions

[pre_execute](#) – Called before a package is installed
[install_mkdirs](#) – Creates directories
[install_copy](#) – Copies files or directories
[install_images](#) – Install images into the custom directory
[install_menus](#) – Installs menus to a specific page or the entire Sugar application
[install_userpage](#) – Adds a section to the User page
[install_dashlets](#) – Installs dashlets into the Sugar application
[install_administration](#) – Installs an administration section into the Admin page
[install_connectors](#) – Installs Sugar Cloud Connectors
[install_vardefs](#) – Modifies existing vardefs
[install_layoutdefs](#) – Modifies existing layouts
[install_layoutfields](#) – Adds custom fields
[install_relationships](#) – Adds relationships
[install_languages](#) – Installs language files
[install_logichooks](#) – Installs a new logic hook
[post_execute](#) – Called after a package is installed

Sample Manifest

The following sample manifest file defines the *\$manifest* and *\$installdef* array elements for a new module (Songs) which depends on two other modules: "Whale Pod" and "Maps". In addition to defining the *\$manifest* and *\$installdef* array elements, it also defines the *\$upgrade_manifest* array.

```
<?php
$manifest = array(
'acceptable_sugar_versions' => array(),
'is_uninstallable' => true,
'name' => 'Song Module',
'description' => 'A Module for all your song needs',
'author' => 'Ajay',
'published_date' => '2005/08/11',
'version' => '2.0',
'type' => 'module',
'icon' => "",
'dependencies' => array (
array ('id_name' => 'whale_pod', 'version => '1.0'),
array ('id_name' => 'maps', 'version => '1.5'),
),
);
$installdefs = array (
'id' => 'songs',
'image_dir' => '<basepath>/images',
'copy' => array (
array (
'from' => '<basepath>/module/Songs',
'to' => 'modules/Songs',
),
),
);
```



```

),
'language' => array (
'from'=> '<basepath>/administration/en_us.songsadmin.php',
'to_module'=> 'application',
'language'=>'en_us'
),
array (
'from'=> '<basepath>/administration/en_us.songsadmin.php',
'to_module'=> 'Administration',
'language'=>'en_us'
),
),
),
'layoutdefs'=> array(
array(
'from'=> '<basepath>/layoutdefs/contacts_layout_defs.php',
'to_module'=> 'Contacts',
),
),
'layoutfields'=> array(
array(
'additional_fields'=> array(
'Songs' => 'music_name_c',
),
),
array(
'additional_fields'=> array(
'Songs' => 'label_company_c',
),
),
),
-
),
'vardefs'=> array(
array(
'from'=> '<basepath>/vardefs/contacts_vardefs.php',
'to_module'=> 'Contacts',
),
),
'administration'=> array (
array ( 'from'=>'<basepath>/administration/songsadminoption.php', ),
),
'beans'=> array (
array (
'module'=> 'Songs',
'class'=> 'Song',
'path'=> 'modules/Songs/Song.php',
'tab'=> true,
)
),
'relationships'=>array (
array (
'meta_data'=>'<basepath>/relationships/contacts_songsMetaData.php',
),
array (
'meta_data'=>'<basepath>/relationships/products_songsMetaData.php',
),
),

```



```

)
),
'custom_fields'=>array (
  array (
    'name'=> 'music_name',
    'label'=>'Music Name',
    'type'=>'varchar',
    'max_size'=>255,
    'require_option'=>'optional',
    'default_value'=>' ',
    'ext1' => 'name',
    'ext2' => 'Accounts',
    'ext3' => ' ',
    'audited'=>0,
    'module'=>'Songs',
  ),
  array (
    'name'=>'label_company',
    'label'=>'Label',
    'type'=>'relate',
    'max_size'=>36,
    'require_option'=>'optional',
    'default_value'=>' ',
    'ext1'=>'name',
    'ext2'=>'Accounts',
    'ext3'=>' ', 'audited'=>0,
    'module'=>'Songs',
  ),
),
),
);

-
$upgrade_manifest = array (
  'upgrade_paths' => array (
    '1.0' => array(
      'id'=>'songs',
      'copy'=>array(
        array (
          'from'=> '<basepath>/module/Songs',
          'to'=> 'modules/Songs',
        ),
      ),
    ),
    '1.5' => array (
      'id'=>'songs',
      'copy' => array(
        array (
          'from'=> '<basepath>/module/Songs',
          'to'=> 'modules/Songs',
        ),
      ),
    ),
  ),
);
?>

```



Business Logic Hooks

[Custom Logic](#) (or "Business Logic Hooks") allows you to add functionality to certain actions, such as [before saving a bean](#), in an upgrade-safe manner. This is accomplished by defining hooks, which are placed in the `custom/` directory, which will be called in the SugarBean in response to events at runtime. Because the code is located separate from the core SugarCRM code, it is upgrade-safe. See [this SugarForge project](#) for a complete example of a useful business hook.

Hook Definition

The code that declares your custom logic is located in: `custom/modules/<CURRENT_MODULE>/logic_hooks.php`. The format of the declaration follows.

[\\$hook_version](#)

All logic hooks should define the `$hook_version` that should be used. Currently, the only supported `$hook_version` is 1.

```
$hook_version = 1
```

[\\$hook_array](#)

Your logic hook will also define the `$hook_array`. `$hook_array` is a two dimensional array:

- o `name`: the name of the event that you are hooking your custom logic to
- o `array`: an array containing the parameters needed to fire the hook

[A best practice is for each entry in the top level array to be defined on a single line to make it easier to automatically replace this file. Also, logic_hooks.php should contain only hook definitions; because the actual logic is defined elsewhere.](#)

For example:

```
$hook_array['before_save'][] = Array(1, test, 'custom/modules/Leads/test12.php', 'TestClass', 'lead_before_save_1');
```

[Available Hooks](#)

The hooks are processed in the order in which they are added to the array. The first dimension is simply the current action, for example **before_save**. The following hooks are available:

[Application hooks](#)

These hooks do not make use of the `$bean` argument.

- o [after_ui_frame](#)
 - o [Fired after the frame has been invoked and before the footer has been invoked](#)
- o [after_ui_footer](#)
 - o [Fired after the footer has been invoked](#)
- o [server_round_trip](#)
 - o [Fired at the end of every Sugar page](#)



Module hooks

- o [before_delete](#)
 - o [Fired before a record is deleted](#)
- o [after_delete](#)
 - o [Fired after a record is deleted](#)
- o [before_restore](#)
 - o [Fired before a record is undeleted](#)
- o [after_restore](#)
 - o [Fired after a record is undeleted](#)
- o [after_retrieve](#)
 - o [Fired after a record has been retrieved from the database. This hook does not fire when you create a new record.](#)
- o [before_save](#)
 - [Fired before a record is saved.](#)
 - [Note: With certain modules, like Cases and Bugs, the human-readable ID of the record \(like the case_number field in the Case module\), is not available within a before_save call. This is because this business logic has not been executed yet.](#)
- o [after_save](#)
 - [Fired after a record is saved.](#)
 - [Note: With certain modules, like Cases and Bugs, the human-readable ID of the record \(like the case_number field in the Case module\), is not available within a before_save call. This is because this business logic simply has not been executed yet.](#)
- o [process_record](#)



- o [Fired immediately prior to the database query resulting in a record being made current. This gives developers an opportunity to examine and tailor the underlying queries. This is also a perfect place to set values in a record's fields prior to display in the DetailView or ListView. This event is not fired in the EditView.](#)

Hooks for Users module

- o [*before_logout*](#)
 - o [Fired before a user logs out of the system](#)
- o [*after_logout*](#)
 - o [Fired after a user logs out of the system](#)
- o [*after_login*](#)
 - o [Fired after a user logs into the system.](#)
- o [*after_logout*](#)
 - o [Fired after a user logs out of the system.](#)
- o [*before_logout*](#)
 - o [Fired before a user logs out of the system.](#)
- o [*login_failed*](#)
 - o [Fired on a failed login attempt](#)

Options Array

[The second dimension is an array consisting of:](#)

- o [Processing index => For sorting before exporting the array](#)
- o [Label/type => Some string value to identify the hook](#)



- o [PHP file to include => Where your class is located. Insert into ./custom.](#)
- o [PHP class the method is in => The name of your class](#)
- o [PHP method to call => The name of your method](#)

[Thus a given class may have multiple methods to fire, each within a single upgrade-safe PHP file. The method signature for version 1 hooks is:](#)

```
function NAME(&$bean, $event, $arguments)
```

[Where:](#)

- o **[\\$bean](#)** - \$this bean passed in by reference.
- o **[\\$event](#)** - The string for the current event (i.e. before_save)
- o **[\\$arguments](#)** - An array of arguments that are specific to the event.

[Packaging Custom Logic Hooks](#)

[You can package logic hooks with a package and load through the Module Loader.](#)

[Along with the other directives for packaging files you can include an entry for 'logic_hooks'. A snippet of the manifest.php may look similar to this:](#)

```
<?php
$installdefs = array(
'logic_hooks' => array(
array(
'module' => 'Accounts',
'hook' => 'after_save',
'order' => 99,
'description' => 'Account sample logic hook',
'file' => 'modules/Sample/sample_account_logic_hook_file.php',
'class' => 'SampleLogicClass',
'function' => 'accountAfterSave',
),
),
);
```

[You must include the 'modules/Sample/sample_account_logic_hook_file.php' as part of your install_defs 'copy' directive so it can be run from the logic hook. When the Module Loader encounters the 'logic_hooks' entry in the installdefs, it will write out the appropriate file so that your logic hooks can be executed.](#)

[Using Custom Logic Hooks](#)

[The section above shows how to create a custom logic hook that runs the function updateDescription\(\) from the class updateDescription \(those do not have to be the same as they are in this example\) in the PHP file updateDescription.php. Below is the actual script from that PHP file.](#)

```
class updateDescription {
function updateDescription(&$bean, $event, $arguments) {
```



```

$bean->description = html_entity_decode($bean->description);
$bean->fetched_row['description'] = $bean->description;
}
}

```

You see that the \$bean is fed into the function and allows access to all of the fields for the currently selected record. Any changes you make to the array will be reflected in the actual data. For example, in this script we are changing the description field. As shown above, there is \$event is set to whatever event is currently running like after_retrieve or before_delete.

Tips

A few cautions around using logic hooks apply:

- o [It is not possible to hook logic to beans being displayed in a sub-panel.](#)
- o [There is no bean that fires specifically for the ListView, DetailView or EditViews user interface events. Use either the 'process_record' or 'after_retrieve' logic hooks.](#)
- o [In order to compare new values with previous values to see whether a change has happened, use \\$bean->fetched_row\['<field>'\] \(old value\) and \\$bean-><field> \(new value\) in the before_save logic hook.](#)

- o [Make sure that the permissions on your logic_hooks.php file and the class file that it references are readable by the web server. If this is not done, Sugar will not read the files and your business logic extensions will not work.](#)
- o [For example, *nix developers who are extending the Tasks logic should use the following command for the logic_hooks file and the same command for the class file that will be called.](#)

```
[sugarcrm]# chmod +r custom/modules/Tasks/logic_hooks.php
```

- o [Make sure that the entire custom/ directory is writable by the web server, or else the logic hooks code will not work properly.](#)

User Interface Customizations

[While the intention of the metadata framework is to abstract some of the value retrieval and display logic for the modules, there will inevitably be a need to customize the framework with separate hooks and logic pertaining to the module's unique business needs.](#)

Custom Grouping of Values

[This is a common scenario in DetailViews, especially for address blocks. This can be achieved as follows:](#)

```

array (
  'name' => 'date_modified',
  'customCode' => '{$fields.date_modified.value} {$APP.LBL_BY} {$fields.modified_by_name.value}',
  'label' => 'LBL_DATE_MODIFIED',
),

```

[This will group the date_modified value and the modified_by_name values together with the \\$APP.LBL_BY label in between. The 'customCode' key is a direct Smarty inline code replacement. At the time of parsing the \\$fields array will be populated with the values populated for the request bean instance.](#)



Custom Buttons

By default, the metadata framework provides default implementations for Cancel, Delete, Duplicate, Edit, Find Duplicates, and Save buttons. However, you may wish to use some of the default buttons or add additional buttons. Here is an example:

```
'templateMeta' => array(
  'form' => array(
    'buttons'=>array('EDIT', 'DUPLICATE', 'DELETE', array(
      'customCode'=>
        '<form action="index.php" method="POST" name="Quote2Opp" id="form">
          <input title="{ $APP.LBL_QUOTE_TO_OPPORTUNITY_TITLE}"
            accessKey="{ $APP.LBL_QUOTE_TO_OPPORTUNITY_KEY}"
            class="button" type="submit"
            name="opp_to_quote_button"
            value="{ $APP.LBL_QUOTE_TO_OPPORTUNITY_LABEL}">
          </form>'
        )
      )
  )
)
```

Here we are adding a custom button with the label defined in the Smarty variable `{ $APP.LBL_QUOTE_TO_OPPORTUNITY_LABEL }`. The EDIT, DUPLICATE and DELETE buttons are rendered and then the snippet of code in this 'customCode' block is added. The code to render the buttons specified in the metadata files may be found in `include/Smarty/function.sugar_button.php`.

Creating New Custom Displays

In this scenario, you wish to take advantage of the metadata framework to do some of the processing for a subset of the fields. However, the provided user interface generated does not meet the needs of your module, which requires richer UI functionality. Typically this scenario occurs when the EditView or DetailView for the module contains UI components that do not fit the framework of a table layout that produced by the metadata. For example, in addition to displaying the properties for the bean instance of the module, there is also a lot of information to be displayed for related beans, mashups, etc. This type of customization may be achieved by a combination of overriding the footer.tpl file in the templateMeta section of the metadata file and creating a view.edit.php file to override the default MVC EditView handling. For example, consider the Quotes module's editviewdefs.php file:

```
'templateMeta' => array(
  'maxColumns' => '2',
  'widths' => array(
    array('label' => '10', 'field' => '30'),
    array('label' => '10', 'field' => '30')
  ),
  'form' => array('footerTpl' => 'modules/Quotes/tpls/EditViewFooter.tpl'),
)
```

Note: You do not have to necessarily create a view.edit.php file, but usually at this point of customization, you will want to add variables to your customized template that are not assigned by the generic EditView handling from the MVC framework. See the next example, Overriding the View, for more information about sub-classing the EditView and assigning Smarty variables to the resulting templates.

Your EditViewFooter.tpl file can now render the necessary user interface code that the generic metadata framework could not:

```
{ $SOME_CRAZY_UI_WIDGET } <----- You can either create this HTML in edit.view.php or place it here
```



```
<applet codebase="something"> <---- Let's add an Applet!
```

```
</applet>
```

EOF expected: /content/body/div[8]/div[7]/div[40]/div[16]/a/span, line 1, column 9 (click for details)

```
<--- Include the generic footer.tpl file at the end
```

[If you want to edit the top panel to provide customized widgets then you can override the header.tpl file instead. In that scenario, the smarty tag to include the generic header.tpl would likely appear at the top of the custom template file.](#)

Overriding the View

[Using the MVC framework, you define your module's own view.edit.php or view.detail.php file to subclass ViewEdit \(for EditView\) or ViewDetail \(for DetailView\). Then you override either the process or display methods to do any intermediary processing as necessary. This technique should be employed when you still want to take advantage of the rendering/layout formatting done by the metadata framework, but wish to tweak how some of the values are retrieved.](#)

```
// Contents of module/[module]/view/view.edit.php file
require_once('include/json_config.php');
require_once('include/MVC/View/views/view.edit.php');
class CallsViewEdit extends ViewEdit {
function CallsViewEdit(){
parent::ViewEdit();
}

-
function display() {
global $json;
$json = getJSONObj();
$json_config = new json_config();
if (isset($this->bean->json_id) && !empty($this->bean->json_id)) {
$JavaScript = $json_config->get_static_json_server(false, true, 'Calls', $this->bean->json_id);
} else {
$this->bean->json_id = $this->bean->id;
$JavaScript = $json_config->get_static_json_server(false, true, 'Calls', $this->bean->id);
}
}

-
// Assign the Javascript code to Smarty .tpl
$this->ss->assign('JSON_CONFIG_JAVASCRIPT', $JavaScript);
parent::display();
}
}
```

[In the file modules/Calls/metadata/editviewdefs.php we have the following defined for the templateMeta->JavaScript value:](#)

```
'JavaScript' => '<script type="text/JavaScript">{$JSON_CONFIG_JAVASCRIPT}</script>'
```

[Here the \\$JSON_CONFIG_JAVASCRIPT Smarty variable was the result of a complex operation and not available via the vardefs.php file \(i.e. there is no JSON_CONFIG_JAVASCRIPT declaration in the vardefs.php file\).](#)



Creating a Custom Sugar Field

Let's try to create a new type of field for rendering a YouTube video. In this example, we will use a custom text field in the Contacts module and then override the Detail View of the custom field in the metadata file to link to our YouTube video.

The process is as follows:

- 1) [Create a custom text field in the Contacts module from the Studio editor](#)
- 2) [Add this custom text field to both the Edit View and Detail View layouts](#)
- 3) [Save and deploy both the updated layouts.](#)
- 4) [Create a directory `include/SugarFields/Fields/YouTube`](#). The name of the directory (YouTube) corresponds to the name of the field type you are creating.
- 5) [In `include/SugarFields/Fields/YouTube` directory, create the file `DetailView.tpl`](#). For the `DetailView` we will use the "embed" tag to display the video. In order to do this, you need to add the following to the template file:

```
{if !empty(  
EOF expected: /content/body/div[9]/div[8]/a/span, line 1, column 10 (click for details)  
)}  
<object width="425" height="350">  
<param name="movie" value="http://www.youtube.com/v/  
EOF expected: /content/body/div[9]/div[10]/a/span, line 1, column 10 (click for details)  
></param>  
<param name="wmode" value="transparent"></param>  
<embed src="http://www.youtube.com/v/  
EOF expected: /content/body/div[9]/div[12]/a/span, line 1, column 10 (click for details)  
" type="application/x-shockwave-flash"  
wmode="transparent" width="425" height="350">  
</embed>  
</object>  
{/if}
```

You will notice that we use the "{{" and "}}" double brackets around our variables. This implies that that section should be evaluated when we are creating the cache file. Remember that Smarty is used to generate the cached templates so we need the double brackets to distinguish between the stage for generating the template file and the stage for processing the runtime view.

Also note that will use the default `EditView` implementation that is provided by the base sugar field. This will give us a text field where people can input the YouTube video ID, so you do not need to create `EditView.tpl`. Also, we do not need to provide a PHP file to handle the `SugarField` processing since the defaults will suffice.

- 6) [Now go to `custom/modules/Contacts/metadata/detailview.php` and add a type override to your YouTube field and save. In this example, the custom field is named "youtube".](#)

```
array (  
'name' => 'youtube_c',
```



```
'type' => 'YouTube',  
'label' => 'LBL_YOUTUBE',  
),
```

Your custom field is now ready to be displayed. You can now find the ID value of a YouTube video to insert into the EditView, and render the video in the DetailView. Remember to set your system to Developer Mode, or delete the EditView.tpl or DetailView.tpl files in the cache/modules/Contacts directory.

Adding QuickSearch to a Custom Field

1. [Include the default configs from QuickSearchDefaults.php. Most of the time you can use the predefined configurations and scripts.](#)

```
require_once('include/QuickSearchDefaults.php');  
$qsd = new QuickSearchDefaults();
```

2. [Then set up the config for the input box you wish to have SQS. Account, Team, and User configs are available. The following configures SQS for account search on input box w/ id of 'parent_name', user and team in similar fashion with defaults.](#)

```
$sqs_objects = array('parent_name' => $qsd->getQSParent(),  
'assigned_user_name' => $qsd->getQSUser(),  
'team_name' => $qsd->getQSTeam());
```

[Notes on structure of config](#) - replace the default parameters if they are different for the page.

- o **method**: Unless you make a new method on the JSON server, keep this as query.
- o **populate_list**: This defines the id's of the fields to be populated after a selection is made.
- o QuickSearch will map the first item of field_list to the first item of populate_list. ie. field_list[0] = populate_list[0], field_list[1] = populate_list[1].... until the end of populate list.
- o **limit**: reduce from 30 if query is large hit, but never less than 12.
- o **conditions**: options are like_custom, contains, or default of starts with
if using 'like_custom' also define 'begin'/'end' for strings to prepend or append to the user input
- o **disable**: set this to true to disable SQS (optional, useful for disabling SQS on parent types in calls for example)
- o **post_onblur_function**: this is an optional function to be called after the user has made a selection. It will be passed in an array with the items in field_list as keys and their corresponding values for the selection as values.

```
$sqs_objects = array('account_name' => // this is the id  
array(
```




```
// the method on to use on the JSON server
'method' => 'query',
'modules' => array('Accounts'), // modules to use
'field_list' => array('name', 'id'), // columns to select
// id's of the html tags to populate with the columns.
'populate_list' => array('account_name', 'account_id'),
'conditions' => // where clause, this code is for any account names that have A WORD
beginning with ...
array(array('name'=>'name','op'=>'like_custom','end'=>'%','value'=>')),
array('name'=>'name','op'=>'like_custom','begin'=>'% ','end'=>'%','value'=>')),
'group' => 'or', // grouping of the where conditions
'order' => 'name', // ordering
'limit' => '30', // number of records to pull
'no_match_text' => $app_strings['ERR_SQS_NO_MATCH'] // text for no matching results
),
```

3. [Include the necessary javascript files if sugar_grp1.js is not already loaded on the page.](#)

```
$quicksearch_js = '<script type="text/javascript" src="" . getJSPath('include/javascript/
sugar_grp1.js') . ""></script>';
```

4. [Assign your config array to **sqs_objects** \(important!\)](#)

```
$quicksearch_js .= '<script type="text/javascript" language="javascript">
sqs_objects = ' . $json->encode($sqs_objects) . '</script>';
```

5. [Add validation so that if there is no matching id for what the user has entered in an SQS field, an alert shows. This is for fields such as assigned users where the form must submit a valid ID.](#)

```
$javascript->addToValidateBinaryDependency('account_name', 'alpha',
app_strings['ERR_SQS_NO_MATCH_FIELD'] .
$mod_strings['LBL_MEMBER_OF'], 'false', "", 'parent_id');
```

6. [Add id tags and class name to the input box. Note that the input box must have class="sqsEnabled"!](#)

```
<input class="sqsEnabled" id="account_name" name='account_name' size='30' type='text'
value="{ACCOUNT_NAME}">
<input id='account_id' name='account_id' type="hidden" value='{ACCOUNT_ID}'>
```

[Having trouble? Take a look at the file *module/Contacts/BusinessCard.php*.](#)

[Removing Downloads Tab for Sugar Plug-ins](#)

[Sugar Enterprise and Professional editions now include a Downloads tab that appears on the User Preferences page. This tab lists links to download Sugar Plug-ins for Microsoft Office. If you do not want to display the Downloads tab to administrators and users, you need to add a configuration parameter named `disable_download_tab` to the `config_override.php` file with the value set to true, as shown below.](#)

```
$sugar_config['disable_download_tab'] = true;
```



Tips

This section provides some tips you will find useful in using the new MVC and metadata framework and to avoid pitfalls that you may run into from the subtle details of the metadata framework that sometimes cause frustration and confusion. Here are some common scenarios that you may have to address.

Grouping Required Fields Together

To group all required fields together all you need to do is set the config to:

```
$GLOBALS['sugar_config']['forms']['requireFirst'] = true;
```

Displaying data on EditViews with a read-only ACL setting

```
$GLOBALS['sugar_config']['showDetailData'] = true;
```

The field value specified in metadata does not appear

This usually happens when the user is attempting to specify a variable name in the module's class definition, but not in the vardefs.php file. For example, consider the Products module. In the Products.php class file there is the variable:

```
var $quote_name;
```

If you attempt to retrieve the value in your metadata file as follows:

```
...  
array (  
  'quote_name',  
  'date_purchased',  
)  
...
```

You must also make sure that 'quote_name' is specified in the vardefs.php file:

```
'quote_name' =>  
array (  
  'name' => 'quote_name',  
  'type' => 'varchar',  
  'vname' => 'LBL_QUOTE_NAME',  
  'source' => 'non-db',  
  'comment' => 'Quote Name'  
)
```

This is because the metadata framework populates the Smarty variable \$fields using the variables listed in the vardefs.php file. If they are not listed, there is no way for the metadata framework to know for sure which class variables have been set and are to be used. The metadata framework works in conjunction with ACL (Access Control Level) checks in the system and these are tied to fields defined in the vardefs.php file.

The field value specified in metadata does not appear but is in vardefs.php

This may occur if the variable in the module has not been initialized. The metadata framework invokes the fill_in_additional_detail_fields() method so be sure the values are either set in the constructor or in the fill_in_additional_detail_fields() method.

A good strategy to debug the Smarty templates that are generated via the metadata files is to check the cache/modules/<module> directory for the .tpl files (DetailView.tpl, EditView.tpl, etc.). Enable the Developer Mode setting under the Advanced panel found through the Admin ->System Settings link to allow for the Smarty templates to be regenerated on every request.

Creating New Sugar Dashlets

A Module View is the simplest Sugar Dashlet to create. This is a customizable ListView of a Sugar Dashlet. For this section we will use the MyAccountsDashlet as an example.



```

MyAccountsDashlet.php:
// include the base class
require_once('include/Dashlets/DashletGeneric.php');
// required for a seed bean
require_once('modules/Accounts/Account.php');

-
class MyAccountsDashlet extends DashletGeneric {
// takes an $id, and $def that contains the options/title/etc.
function MyAccountsDashlet($id, $def = null) {
require_once('MyAccountsDashlet.data.php');
parent::DashletGeneric($id, $def);
global $current_user, $app_strings;
$this->searchFields =
$DashletData['MyAccountsDashlet']['searchFields'];
$this->columns =
$DashletData['MyAccountsDashlet']['columns'];
// define a default title
if(empty($def['title'])) $this->title = 'My Account Dashlet';
$this->seedBean = new Account();
}
}

```

All the metadata for this Sugar Dashlet is defined in the constructor. `$searchFields` are the search inputs that can be applied to the view. Defining these here will tell which input fields to generate corresponding filters when the user configures the Sugar Dashlet. `$columns` define the available columns to the user. These contain the visible columns and the columns the user can make visible. Both columns and searchFields are defined in `MyAccountsDashlet.data.php` so that Studio can modify them easily.

A seed bean is also required.

```

MyAccountsDashlet.data.php:
$DashletData['MyAccountsDashlet']['searchFields'] =
array('date_entered' => array('default' => ''));

-
$DashletData['MyAccountsDashlet']['columns'] = array(
'name' => array(
'width' => '40',
'label' => 'LBL_LIST_ACCOUNT_NAME',
'link' => true, // is the column clickable
'default' => true // is this column displayed by default
),
'billing_address_state' => array(
'width' => '8',
'label' => 'LBL BILLING ADDRESS STATE')
);

```

This file along with the `MyAccountsDashlet.meta.php` file is enough to create a generic module view Sugar Dashlet (see below).

Custom Sugar Dashlets

Sugar Dashlets are more than generic module views. They can provide unlimited functionality and integration.

For this section we will use the JotPad Sugar Dashlet as an example. The JotPad is a simple note taking Sugar Dashlet. A user double clicks on the Sugar Dashlet and can enter any text in the Sugar Dashlet. When the user clicks outside of the textarea, the text is automatically saved via AJAX.



[There are six files that define this Sugar Dashlet](#)

1. [JotPadDashlet.php – JotPad Class](#)
2. [JotPadDashlet.meta.php – metadata about the Sugar Dashlet](#)
3. [JotPadDashlet.tpl – Display Template](#)
4. [JotPadDashletOptions.tpl – Configuration template](#)
5. [JotPadDashletScript.tpl - Javascript](#)
6. [JotPadDashlet.en_us.lang.php – English Language file](#)

[JotPadDashlet.php:](#)

```
// this extends Dashlet instead of DashletGeneric
class Dashlet extends Dashlet {
    var $savedText; // users's saved text
    var $height = '100'; // height of the pad

    -
    function JotPadDashlet($id, $def) {
        $this->loadLanguage('JotPadDashlet'); // load the language strings

        -
        // load default text is none is defined
        if(!empty($def['savedText']))
            $this->savedText = $def['savedText'];
        else
            $this->savedText = $this->DashletStrings['LBL_DEFAULT_TEXT'];
        // set a default height if none is set
        if(!empty($def['height']))
            $this->height = $def['height'];

        -
        // call parent constructor
        parent::Dashlet($id);
        // Dashlet is configurable
        $this->isConfigurable = true;
        // Dashlet has JavaScript attached to it
        $this->hasScript = true;
        // if no custom title, use default
        if(empty($def['title']))
            $this->title = $this->DashletStrings['LBL_TITLE'];
        else
            $this->title = $def['title'];
    }

    -
    // Displays the Dashlet
    function display() {}
    // Displays the JavaScript for the Dashlet
    function displayScript() {}
    // Displays the configuration form for the Dashlet
```



```

function displayOptions() {}

-
// called to filter out $ _REQUEST object when the
// user submits the configure dropdown
function saveOptions($req) {}

-
// Used to save text on textarea blur.
// Accessed via Home/CallMethodDashlet.php
function saveText() {}
}

```

JotPadDashletOptions.tpl:

```

<form name='configure_{ $id}' action='index.php' method='post' onSubmit='return
SUGAR.Dashlets.postForm("configure_{ $id}", SUGAR.sugarHome.uncoverPage);'>

```

The important thing to note here is the onSubmit. All configure forms should have this statement to uncover the page to remove the configuration dialog.

NOTE: It is important to separate your JavaScript into a separate JavaScript file. This is because Sugar Dashlets are dynamically added to a page through AJAX. The HTML included into JavaScript is not evaluated when dynamically included.

It is important that all JavaScript functions are included in this script file. Inline JavaScript (<a href onclick="" etc) will still function. If the Sugar Dashlet has JavaScript and a user dynamically adds it to the page, the Sugar Dashlet will not be accessible until after the user reloads the page.

Therefore it is beneficial to use as many generic methods in Dashlet.js as possible (Dashlets.callMethod() specifically!).

JotPadDashletScripts.tpl:

```

{literal}<script>
// since the Dashlet can be included multiple times a page,
// don't redefine these functions
if(typeof JotPad == 'undefined') {
JotPad = function() {
return {
blur: function(ta, id) {}, // called when textarea is blurred
edit: function(divObj, id) {}, // called when textarea is dbl clicked
saved: function(data) {}, // callback for saving
};
};
}
</script>{/literal}

```

Please refer to the file for more detail comments.

Packaging Custom Sugar Dashlets

To make a Sugar Dashlet Module installable, you will need to package with the following also included in the manifest.php files.

```

$installdefs = array( 'id'=> 'jotpad', 'Dashlets'=> array( array('name' => 'JotPad', 'from' =>
'<basepath>/JotPad', ), ), );

```

Refreshing the Sugar Dashlet Cache

To add a Sugar Dashlet to your SugarCRM installation, you can use the Module Loader to install your Sugar Dashlet Package. However, for development purposes, to make the Sugar Dashlet available to add to the home page you will need to run the Repair Sugar Dashlet Link in the Admin Repair Panel at least once after you have created the Dashlet.

This will rebuild the cache file /<cache dir>/Dashlets/Dashlets.php by scanning the folders /modules/ and /custom/modules/ for Dashlet Files.

Creating Custom Chart Dashlets



Creating a custom chart dashlet is very similar to how we created the [MyAccountsDashlet](#) above. The main difference is that you will need to override the `display()` method in your class to build the chart, using the SugarChart library included with SugarCRM. Below is an example `display()` method as used in the Outcome by Month dashlet.

```

public function display()
{
    $currency_symbol = $GLOBALS['sugar_config']['default_currency_symbol'];
    if ($GLOBALS['current_user']->getPreference('currency')){
        require_once('modules/Currencies/Currency.php');
        $currency = new Currency();
        $currency->retrieve($GLOBALS['current_user']->getPreference('currency'));
        $currency_symbol = $currency->symbol;
    }

    -
    require("modules/Charts/chartdefs.php");
    $chartDef = $chartDefs['outcome_by_month'];
    require_once('include/SugarCharts/SugarChart.php');
    $sugarChart = new SugarChart();
    $sugarChart->setProperties(",
    translate('LBL_OPP_SIZE', 'Charts') . ' ' . $currency_symbol . '1' . translate('LBL_OPP_THOUSANDS',
    'Charts'),
    $chartDef['chartType']);
    $sugarChart->base_url = $chartDef['base_url'];
    $sugarChart->group_by = $chartDef['groupBy'];
    $sugarChart->url_params = array();
    $sugarChart->getData($this->constructQuery());
    $xmlFile = $sugarChart->getXMLFileName($this->id);
    $sugarChart->saveXMLFile($xmlFile, $sugarChart->generateXML());
    return $this->getTitle('<div align="center"></div>') .
    '<div align="center">' . $sugarChart->display($this->id, $xmlFile, '100%', '480', false) . '</div><br
    />';
}

-
protected function constructQuery()
{
    $query = "SELECT sales_stage,".
    db_convert('opportunities.date_closed','date_format',array("%Y-%m"),array("YYYY-MM"))." as m,".
    ".
    "sum(amount_usdollar/1000) as total, count(*) as opp_count FROM opportunities ";
    $query .= " WHERE opportunities.date_closed >= ".db_convert("".$this->obm_date_start.""', 'datetime') .
    " AND opportunities.date_closed <= ".db_convert("".$this->obm_date_end.""', 'datetime') .
    " AND opportunities.deleted=0";
    if (count($this->obm_ids) > 0)
    $query .= " AND opportunities.assigned_user_id IN (" . implode(",",$this->obm_ids) . ")";
    $query .= " GROUP BY sales_stage,".
    db_convert('opportunities.date_closed','date_format',array("%Y-%m"),array("YYYY-MM")) . "
    ORDER BY m";
    return $query;
}

```



Themes

The goal of the Themes framework is to reduce the complexity of the current themes and the amount of work needed to create new themes in the product. The framework also provides tools for easily changing a theme in an upgrade-safe way without modifying the theme directly. This is all possible through an inheritance mechanism that is part of the framework, where themes can build upon other themes to build the user interface for the application. This directly reduces the duplication of code in the application, and enables new UI elements to be supported more easily in any theme.

Theme Directory Structure

The theme directory has a more formal structure to it, which must be followed in order for the inheritance mechanism in the themes framework to work correctly. It is as follows:

themes/

<theme name>/

themedef.php

css/ // all css files go here

style.css

print.css

images/ // all images go here

js/ // all js files go here

The themedef.php file specified also has a specific format to it so that it can be interpreted properly by the application. It is as follows:

```
$themedef = array(
```

```
'name' => "Sugar", // theme name
```

```
'description' => "Sugar", // short description of the theme
```

```
'maxTabs' => $max_tabs, // maximum number of tabs shown in the bar
```

```
'pngSupport' => true, // true if png image files are used in this theme, false if gifs
```

```
'parentTheme' => "ParentTheme", // name of the theme this theme inherits from, if something other than the default theme.
```

```
'barChartColors' => array(...),
```

```
'pieChartColors' => array(...),
```

```
);
```

Please note that only the 'name' specification is required; all other elements are optional.

Theme Development

When you are developing a new theme or adding modifications to an existing theme, it is recommended to turn developer mode on in the 'System Settings' in the 'Admin' section of the application. This is so that any changes you make will become immediately visible to you, making testing your changes easier. Once the theme development is complete, it is recommended that you turn off the Developer Mode.

The theme framework performance is greatly enhanced with the use of an external caching mechanism such as APC or Memcache. It is highly recommended to have this in place.

Changing a Theme

Modifications to any theme in the application can be made in the custom/themes/ directory under the theme in question. For example, to replace the default Accounts.gif image in the Sugar theme, drop the new.gif image file into the custom/themes/<theme name>/images/ directory. You can do the same for CSS and JS files; in these cases the given files will be appended to the existing ones instead of being used in place of them.

The order in which directories are searched for overriding images/css/js files is as follows:

1. custom/themes/<theme name>/



- 2. themes/<theme name>/
- 3. any parent themes (custom directory first, then regular one)
- 4. custom/themes/default/
- 5. themes/default/

Creating a New Theme

The easiest way to create a new theme is to find a base theme that you like and base your design on it. This reduces the amount of CSS work you'll have to do on your own, and you can rest assured that any theme fixes will also be fixed in your theme, thanks to inheritance. To do this, you'll need to create a `themedef.php` file with the following specifications in it:

```
$themedef = array(
'name' => "MySugar", // theme name
'description' => "Sugar theme for me", // optional, short description of the theme
'parentTheme' => "Sugar", // name of the theme this theme inherits from, in this case the Sugar theme
);
```

You can now add any CSS, images, and/or JS code to their respective directories to make the needed alterations to the parent theme to create the desired theme. It is by no means a requirement to derive a theme from one of the existing ones in the product; if you do not specify the 'parentTheme' attribute above, then the theme settings in the `themes/default/` directory will be used.

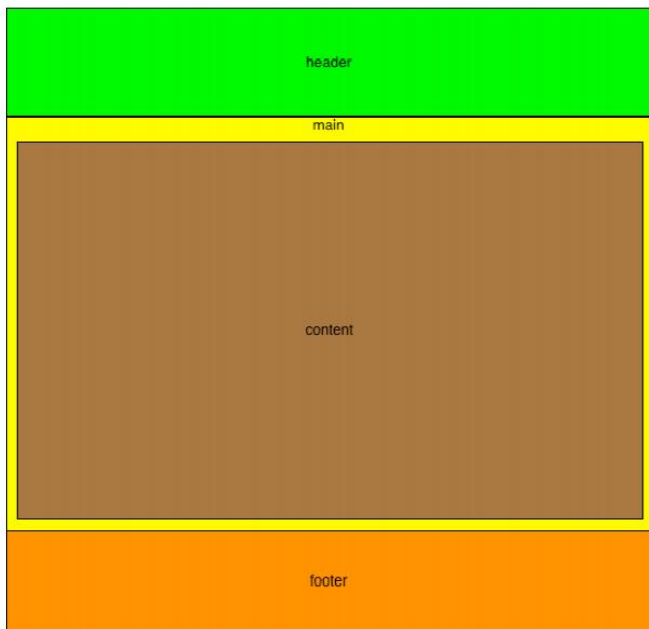
Element Reference Guide

Below is a guide of the various id elements and classes that are used in the application. IDs are indicated below where they are prefixed by a # sign (i.e. #dog for the ID dog), and classes are specified by a '.' (.cat for class cat).

.edit	EditView containers
.detail	DetailView containers
.list	ListView containers
.view	Styles for any of the detail, edit, list, and search view containers
.search	Search container

The below figure illustrates the where the main UI div elements are located and what their IDs are.





Packaging Custom Themes

[The Upgrade Wizard, accessible through the Admin screen, allows you to apply themes without unzipping the files manually. The theme is also actually added to the "Theme" dropdown on the Sugar Login screen.](#)

[The Upgrade Wizard relies on a file named manifest.php, which should reside alongside the root directory of your theme ZIP file.](#)

[The following section outlines the format of the manifest file. An example manifest file can be found in the following section.](#)

- o **[acceptable_sugar_flavors](#)** Contains which Sugar Editions the package can be installed on. Accepted values are any combination of: CE, PRO, and ENT.
- o **[acceptable_sugar_versions](#)** This directive contains two arrays:
 - o [exact_matches: each element in this array should be one exact version string, i.e. "6.0.0b" or "6.1.0"](#)
 - o [regex_matches: each element in this array should be one regular expression designed to match a group of versions, i.e. "6\\.1\\.0\[a-z\]"](#)
- o **[author](#)** Contains the author of the package; for example, "SugarCRM Inc."
- o **[copy_files](#)** An array detailing the source and destination of files that should be copied during installation of the package. See the example manifest file below for details.



- o ***description*** A description of the package. Displayed during installation.
- o ***icon*** A path (within the package ZIP file) to an icon image that will be displayed during installation. Examples include `"/patch_directory/icon.gif"` and `"/patch_directory/images/theme.gif"`
- o ***is_uninstallable*** Setting this directive to TRUE allows the Sugar administrator to uninstall the package. Setting this directive to FALSE disables the uninstall feature.
- o ***name*** The name of the package. Displayed during installation.
- o ***published_date*** The date the package was published. Displayed during installation.
- o ***type*** The package type. This should be set to "theme"
- o ***version*** The version of the patch, i.e. "1.0" or "0.96-pre1"

Example Theme Manifest File

The following is an example `manifest.php` file:

```
<?php
$manifest = array (
  'acceptable_sugar_versions' =>
  array (
    'exact_matches' =>
    array (
      ),
    'regex_matches' =>
    array (
      0 => '3.5.[01][a-z]?'
    ),
  ),
  'acceptable_sugar_flavors' =>
  array (
    0 => 'OS',
    1 => 'PRO',
    2 => 'ENT',
  ),
  'name' => 'Theme Name',
  'description' => 'Theme Description',
  'author' => 'SugarCRM Inc.',
  'published_date' => '2005-09-15 16:00:00',
  'version' => '3.5.1',
  'type' => 'theme',
  'is_uninstallable' => TRUE,
  'icon' => 'ThemeName/images/Themes.gif',
  'copy_files' =>
  array (
    'from_dir' => 'ThemeName',
```



```
'to_dir' => 'themes/ThemeName',  
'force_copy' =>  
array (  
)  
)  
)?>
```

Example File Structure: The following is an example of the file structure of the GoldenGate theme:

```
ThemeName.zip  
| manifest.php | \---ThemeName  
| config.php  
| cookie.js  
| footer.php  
| <more files>  
|  
| \---images  
| accept_inline.gif  
| AccountReports.gif  
| Accounts.gif  
| <more images>
```

You'll need to create a root directory that contains the theme directory. The name of this root directory (ThemeName) is what should be used in the from_dir element of the copy_files array in manifest.php. You'll also need to place your manifest.php alongside this root directory. Create a ZIP file containing the root directory and the manifest.php file at the top level. Now your language pack is ready to be installed with the Upgrade Wizard.

Tips

Pick your Canvas

Think about how you want the general look and feel of your theme, and see which out-of-the-box existing Sugar theme best fits. This will reduce any extra work that may come out of rearranging the layout because you chose the first theme you saw.

Replace All

When changing colors in the css files, do a "replace all" on the file. For example, if you are changing the color '#FF0000' to '#CC0000', change all instances of '#FF0000' to '#CC0000'. Eventually you will get to a point where you may want your changes to only affect one class, and may have to go back and refine some of the mass changes. However doing this initially will usually yield favorable results, and save you some time.

Check your work

So you have all your css laid out and your home page looks good? Did you check your Edit and List views? Did you check the calendar popup widgets (from date fields)? Often, developers forget to check the css for Sugar Dashlets, reports and charts. Do not forget to do a thorough check, and keep in mind that you may have to tweak with navigation.css, layout_utils.php, and sugarColors.xml files before being finally done.



Personalize your theme

Remember this is your theme now. If you want to add a new div, or introduce a new class, you do not have to make it fit within the confines of the theme with which you started. Most of the images are transparent and work fine, but changing the look and feel of those would add an extra degree of customization. So, go ahead and add your flash intro, embedded mp3 or Star Wars Background.

Adding Multiple Languages

Sugar as an application platform is internationalized and localizable. Data is stored and presented in the UTF8 codepage allowing for all character sets to be used. Sugar provides a language pack framework allowing developers to build support for any language to be used in the display of user interface labels. Adding a language is a simple process. Two solutions exist to add your own language to SugarCRM. When you add a new language, you must choose a language prefix. While you can choose any key value you want, we recommend following the standard locale naming convention. For example, en_en for English, en_us for US English, ge_ge for German or ge_ch for Swiss German. This key value will be the prefix for all of your new language files.

Add a Language

You can add a new language directly to Sugar without using a Language Pack. Follow the steps outlined below:

1. Add the new language you are creating to the \$languages variable in the config.php file. For instance, to add Spanish, first modify the config.php file in the root of your Sugar installation to reference the new language pack.

```
$sugar_config['languages'] = array('en_us'=>'US English','es_es'=>'Español');
```

Note: You can also set the default language in the config.php. This value is used if a user does not specify a language while logging into the application.

2. Cut and paste each of the en_us.lang.php string files that ship with Sugar Open Source into a new file (<new>.lang.php) with the new prefix you set in the step above.

There are two general locations where you will need to create the new language files:

- o include/language/<new>.lang.php - This file contains the strings common across the entire application. This file must exist for each language defined in config.php.
- o modules/<some_module>/language/<new>.lang.php - These files contain the strings specific to each module.

Note: Some language files that ship with Sugar are not named "en_us.lang.php" but are ".html" or ".tpl" files. These files are used for the inline help system. A complete language pack will include the translated versions of these files as well.

3. After you create your new files by cutting and pasting the en_us.lang.php files, open each file and translate the strings from English. The strings are defined inside of global arrays and the array syntax does need to be maintained for everything to work. A common problem you will quickly see if the array syntax is off is that the user interface doesn't display at all (just a blank page).



Note: In the `include/language/<new>.lang.php` file, you will see that there are two global arrays defined. The `$app_list_strings` array is actually an array of arrays. The key values in this array must not be changed. There are comments and examples in the code that should keep you on track.

Creating Language Packs

A Language Pack is a compressed file loadable through Module Loader. It is the best solution to add a language to SugarCRM. Indeed, it is easier to maintain and to port to other instances of SugarCRM. By default, the Help link in the Sugar application points to the Sugar Application Guide. If you want the Help link in Sugar to point to localized Help content in a language other than English, you can include your localized inline Help files in the language pack.

There are different ways to create a Language Pack:

- [You can build a Language Pack using one of the tools available on SugarForge like "SugarCRM Translation Module" or "Sugar Translation Suite".](#)
- [You can create it using an existing Language Pack:](#)

- o [Download an existing Language Pack.](#)

Note: Be careful to choose a language pack that is up to date and is working.

- o [Uncompress it and be careful to keep the folder architecture.](#)

- o [Rename all the files with a name starting with the language prefix \(for example "es_es" if you downloaded the Spanish Language Pack\) by replacing the old prefix \("es_es"\) with the prefix of your new language. Be careful to not modify the manifest.php file and the name of the folders.](#)

- o [Open each file that you have renamed and translate the strings from English. The strings are defined inside of global arrays, and the array syntax does need to be maintained for everything to work. A common problem you will see if the array syntax is off is that the user interface does not display at all \(just a blank page\).](#)

- o [Optionally, include localized Help files for each view in each module. For example, for the Portugese \(Brazilian\) language pack, you would include `pt_br.help.index.html` \(for List View\), `pt_br.help.DetailView.html` \(for Detail View\), and `pt_br.help.EditView.html` \(for Edit View\).](#)

- o [Modify the manifest.php file based on the file changes you made.](#)

- o [Compress back everything in a zip file.](#)

- o [You can now load this Language Pack using Module Loader.](#)

- [You can create it from the start to finish:](#)



- o [Follow the steps of "Add a Language" above.](#)
- o [After you ensure that it works on your test instance, package it to be installed.](#)

- o [Module Loader allows you to apply language packs without needing to add the language to the \\$languages array in config.php. The Module Loader relies on a file named manifest.php, which should reside alongside the root directory of your language pack ZIP file.](#)

The following is an example manifest.php file, for the Portugese (Brazilian) language pack:

```
<?php
$manifest = array (
'acceptable_sugar_versions' =>
array (
'exact_matches' =>
array (
),
'regex_matches' =>
array (
0 => '5\.0\.0[a-z]?'
),
),
'acceptable_sugar_flavors' =>
array (
0 => 'CE',
1 => 'PRO',
2 => 'ENT',
),
'name' => 'Portugese (Brazilian) language pack',
'description' => 'Portugese (Brazilian) language pack',
'author' => 'Your name here!',
'published_date' => '2008-07-29 22:50:00',
'version' => '5.0.0',
'type' => 'langpack',
'icon' => '',
'is_uninstallable' => TRUE,
),
$installdefs = array(
'id' => 'pt_br',
'copy' => array(
array(
'from' => '<basepath>/modules',
'to' => 'modules',
),
array(
'from' => '<basepath>/include/language',
'to' => 'include/language'
),
array(
'from' => '<basepath>/install/language',
'to' => 'install/language',
```



```
),  
,  
,:  
-  
?>
```

The following is an example of the file structure of the Portugese (Brazilian) language pack:
[SugarEnt-5.0.0-lang-pt_br-2008-07-29.zip](#)

```
├─  
  └─ manifest.php  
      └─  
          └─ include  
              └─ language  
                  └─ pt\_br.lang.php  
          └─ modules  
              └─ Accounts  
                  └─  
                      └─ language  
                          └─ pt\_br.lang.php  
          └─ Activities  
              └─ language  
                  └─ pt\_br.lang.php  
          └─ <other module directories>
```

You will need to create a root directory that contains the `./include/` and `./modules/` directories of the language pack.

The name of this root directory (i.e. `pt_br_500`) is what should be used in the `from_dir` element of the `copy_files` array in `manifest.php`. You will also need to place your `manifest.php` alongside this root directory.

Then you must copy all the language files that you created as described in "Add a Language" into these directories. Be careful to keep the same directory structure (for examples if the file was in `modules/Accounts/language` copy/paste it in `pt_br_500/modules/Accounts/language`).

Create a Zip file containing the root directory and the `manifest.php` file at the top level. Now your language pack is ready to be installed with the Module Loader.

Creating a Connector

This section describes how to write a connector that can be installed through the Module Loader.

1) [Create project directory and files.](#)

The first step is to create a project directory for your connector. We can call this connector "test" for now and create a directory called "test". Under the "test" directory, also create a sub-directory called "source" and then under the "source" directory, create another sub-directory "language". The "source" directory will contain your connector's code. The minimal files a connector should provide are a config file (`config.php`), a variable definition file (`vardefs.php`) and a connector source file (`test.php` in this case). An optional default mapping file that associates your connector's fields with fields in Sugar's modules may be provided (`mapping.php`). Your directory should look like this:

```
test  
test/source  
test/source/test.php  
test/source/vardefs.php
```



[test/source/config.php](#)
[test/source/mapping.php \(optional\)](#)
-
[test/language](#)
[test/language/en_us.lang.php \(default English language file for localization\)](#)

2) [Create the vardefs.php file.](#)

The next step is to provide a list of variable definitions that your connector uses. These should be the fields that your connector uses. For example, if your connector is a person lookup service, then these fields may be a first and last name, email address and phone number. You must also provide a unique field identifier for each connector record. This unique field needs to be named "id". Each vardef field entry is defined within a PHP Array variable. The syntax is similar to a Sugar module's vardefs.php file with the exception of the 'hidden', 'input', 'search', 'hover' and 'options' keys.

The 'hidden' key/value parameter is used to hide the connector's field in the framework. The required "id" field should be declared hidden because this is a unique record identifier that is used internally by the connector framework.

The 'input' key/value parameter is an optional entry to provide an input argument name conversion. In other words, if your connector service expects a "firstName" argument but your connector's field is named "firstname", you may provide an additional input entry so that the connector framework will call your connector's getItem() and getList() methods with an argument named "firstName". Typically, the 'input' key/value parameter is used to support searching.

```
'lastname' => array (  
  'name' => 'lastname',  
  'vname' => 'LBL_LAST_NAME',  
  'input' => 'lastName',  
  'search' => true,  
  'hover' => 'true',  
)
```

Although the benefit of this is probably not captured well in this example, you could potentially have a service that groups arguments in a nested array. For example imagine the following array of arguments for a connector service:

```
$args['name']['first']  
$args['name']['last']  
$args['phone']['mobile']  
$args['phone']['office']
```

Here we have an array with 'name' and 'phone' indexes in the first level. If your connector expects arguments in this format then you may supply an input key/value entry in your vardefs.php file to do this conversion. The input key value should be delimited with a period (.).

```
'lastname' => array (  
  'name' => 'lastname',  
  'vname' => 'LBL_LAST_NAME',  
  'input' => 'name.last', // Creates Array argument ['name']['last']  
  'search' => true,  
  'hover' => 'true',  
)
```

The 'search' key/value parameter is an optional entry used to specify which connector field(s) are searchable. In step 1 of the connector wizard screen, a search form will be generated for your connector so that the user may optionally refine the list of results shown. Currently, we do not filter the fields that may be added to the search form so the use of the 'search' key/value parameter serves more as a visual indication.



The 'hover' key/value parameter is an optional entry to support the hover functionality. A vardef field that is denoted as the hover field contains the value the hover code will use in displaying a popup that displays additional detail in the Detail Views.

The 'options' key/value parameter allows the developer to map values returned by the connector to values that may be used by the Sugar database. In our example, the state field returns the abbreviated value of the State (CA for California, HI for Hawaii, etc.). If we wish to use the State name instead of the abbreviated name, we may specify the 'options' key/value parameter and then add the mapping entry (mentioned in step 9) to enable this translation. This is especially helpful should your system depend on a predefined set of values that differ from those returned by the connector.

Here is a complete example of our "test" connector's vardefs.php file:

```
<?php
$dictionary['ext_rest_test'] = array(
    'comment' => 'A test connector',
    'fields' => array (
        'id' => array (
            'name' => 'id',
            'vname' => 'LBL_ID',
            'hidden' => true,
        ),
        'firstname' => array (
            'name' => 'firstname',
            'vname' => 'LBL_FIRST_NAME',
        ),
        'lastname' => array(
            'name' => 'lastname',
            'vname' => 'LBL_LAST_NAME',
            'input' => 'name.last',
            'search' => true,
        ),
        'website' => array(
            'name' => 'website',
            'vname' => 'LBL_WEBSITE',
            'hover' => true,
        ),
        'state' => array(
            'name' => 'state',
            'vname' => 'LBL_STATE',
            'options' => 'states_dom',
        ),
    ),
);
?>
```

3) [Create the config.php file.](#)

The config.php file holds a PHP array with two keys. The "name" is used to provide a naming label for your connector that will appear in the tabs throughout the application. The "properties" key may be used to store runtime properties for your connector. Here we have simply provided the name "Test" and a properties value that we may use to control the maximum number of results we return in our connector.

```
<?php
$config = array (
    'name' => 'Test',
```



```
'properties' =>
array (
'max_results' => 50,
),
);
?>
```

4) [Create the Language file contents.](#)

The next step is to create a language file with labels for your application. Notice that the properties defined in the config.php file are indexed by the property key ("max_results"). Otherwise, the vardefs.php entries should be indexed off the "vname" values.

```
<?php
$connector_strings = array (
//vardef labels
'LBL_FIRST_NAME' => 'First Name',
'LBL_LAST_NAME' => 'Last Name',
'LBL_WEBSITE' => 'Website',
'LBL_STATE' => 'State',
//Configuration labels
'max_results' => 'Maximum Number of Results',
);
?>
```

The second step is to determine the connector protocol type to create the connector source file. Currently the two Web Services protocols supported are REST and SOAP. If the web service is a REST protocol, then the connector should extend the ext_rest class defined in the file include/connectors/sources/ext/rest/rest.php. If the web service is a SOAP protocol, then the connector should extend the ext_soap class defined in the file include/connectors/sources/ext/soap/soap.php. In this example, we will extend the ext_rest class. The class name should contain the "ext_rest_" suffix if it is a REST protocol connector or "ext_soap_" if it is a SOAP protocol connector.

```
<?php
require_once('include/connectors/sources/ext/rest/rest.php');
class ext_rest_test extends ext_rest {
-
}
?>
```

6) [Provide implementations for the getItem\(\) and getList\(\) Methods.](#)

There are two methods that a connector must override and provide an implementation for. These are the getItem() and getList() methods. These two methods are called by the component class (include/connectors/component.php). The getList() method as its name suggests, returns a list of results for a given set of search criteria that your connector can handle. On the other hand, the getItem() method should attempt to return a single connector record. For example, if your connector is a person lookup service, the getList() method may return matching person values based on a first and last name search. The getItem() method should return values for a unique person. Your service may uniquely identify a person based on an internal id or perhaps an email address.

The getList() method accepts two arguments. \$args is an Array of argument values and \$module is a String value of the module that the connector framework is interacting with. The getList() method should return a multi-dimensional Array with each record's unique id as the key. The value should be another Array of key/value pairs where the keys are the field names as defined in the vardefs.php file.



```

public function getList($args=array(), $module=null) {
-
- $results = array();
-
- if(!empty($args['name']['last']) && strtolower($args['name']['last']) == 'doe') {
- $results[1] = array('id'=>1, 'firstname'=>'John', 'lastname'=>'Doe',
- 'website'=>'www.johndoe.com', 'state'=>'CA');
- $results[2] = array('id'=>1, 'firstname'=>'Jane', 'lastname'=>'Doe',
- 'website'=>'www.janedoe.com', 'state'=>'HI');
- }
-
- return $results;
-
- }

```

The getItem() method also accepts two arguments. \$args is an Array of argument values and \$module is a String value of the module that the connector framework is interacting with. The getItem() method will be called with a unique id as defined in the getList() method's results.

```

public function getItem($args=array(), $module=null) {
- $result = null;
- if($args['id'] == 1) {
- $result = array();
- $result['id'] = '1'; //Unique record identifier
- $result['firstname'] = 'John';
- $result['lastname'] = 'Doe';
- $result['website'] = 'http://www.johndoe.com';
- $result['state'] = 'CA';
- } else if($args['id'] == 2) {
- $result = array();
- $result['id'] = '2'; //Unique record identifier
- $result['firstname'] = 'Jane';
- $result['lastname'] = 'Doe';
- $result['website'] = 'http://www.janedoe.com';
- $result['state'] = 'HI';
- }
- return $result;
- }

```

7) [Provide optional testing functionality](#)

This is an optional step where you may wish to provide functionality for your connector so that it may be tested through the administration interface under the "Set Connector Properties" section. To enable testing for your connector, set the connector class variable `__has_testing_enabled` to true in the constructor and provide a test () method implementation.

```

public function __construct(){
- parent::__construct();
- $this->__has_testing_enabled = true;
- }
-
- public function test() {
- $item = $this->getItem(array('id'=>'1'));
- return !empty($item['firstname']) && ($item['firstname'] == 'John');
- }

```

8) [Provide optional Hover Link functionality](#)



[This is an optional step where you may wish to provide functionality for your connector so that the DetailView of modules enabled for the connector display a popup with additional information. Depending on the connector field, this step involves creating some additional PHP and Smarty code. A new directory needs to be created to contain the hover code.](#)

[Create a "formatter" sub-directory under the "Test" connector's root folder. Then add the formatter class there. We'll also call this file "test.php". Also create a "tpls" sub-directory under the "formatter" directory. The "tpls" directory may contain an optional icon that will be displayed next to the field in the DetailView that will activate the hover popup. By default, the connector framework uses the icon in themes/default/images/icon_Connectors.gif. The default.tpl file is the Smarty template file that will be activated when the hover icon is launched. Your directory structure should now appear as:](#)

```
test
-
test/source
test/source/test.php
test/source/vardefs.php
test/source/config.php
test/source/mapping.php (optional)
-
test/language
test/language/en_us.lang.php (default English language file for localization)
-
test/formatter
test/formatter/test.php
test/formatter/tpls/test.gif
test/formatter/tpls/default.tpl
```

[The test.php file should extend default formatter \(include/connectors/formatters/default/formatter.php\) with the same prefix naming convention used for the connector class "ext_rest_" in this case. Also, place the prefix "_formatter" after the name of the connector. In our example, we are looking for Sugar module fields that have been mapped to the website connector field. If a module field displayed in the Detail View has been mapped to our connector's website field then we will add the code to display the hover popup.](#)

```
<?php
class ext_rest_test_formatter extends default_formatter {
-
public function getDetailViewFormat() {
    $mapping = $this->getSourceMapping();
    $mapping_name = !empty($mapping['beans'][$this->_module]['website']) ?
    $mapping['beans'][$this->_module]['website'] : "";
-
    if(!empty($mapping_name)) {
        $this->_ss->assign('mapping_name', $mapping_name);
        return $this->fetchSmarty();
    }
-
    $GLOBALS['log']->
    >error($GLOBALS['app_strings']['ERR_MISSING_MAPPING_ENTRY_FORM_MODULE']);
    return "";
}
public function getIconFilePath() {
    return 'custom/modules/Connectors/connectors/formatters/ext/rest/test/tpls/test.jpg';
}
```



```
-  
}  
?>
```

The default.tpl file should contain the code to display additional information. The connector framework will trigger a call to the `show_ext_rest [connector name]` or `show_ext_soap [connector name]` methods depending on the connector type (REST or SOAP) when a mouseover javascript action is detected on the hover field (single icon) or when a mouseclick javascript action is detected on the hover field (multiple hover links shown in dropdown menu).

We provide a simple example below, but typically you may wish to have the hover code make additional calls to retrieve information. For example, you may wish to use an AJAX pattern to make a request back to the Sugar system that will in turn make a REST call to your connector.

```
<div style="visibility:hidden;" id="test_popup_div"></div>  
<script type="text/javascript">  
function show_ext_rest_test(event)  
{  
  literal}  
  {  
-  
  var xCoordinate = event.clientX;  
  var yCoordinate = event.clientY;  
  var isIE = document.all?true:false;  
  if(isIE) {  
    xCoordinate = xCoordinate + document.body.scrollLeft;  
    yCoordinate = yCoordinate + document.body.scrollTop;  
  }  
-  
  }/literal}  
-  
  cd = new CompanyDetailsDialog("test_popup_div", 'This is a test', xCoordinate, yCoordinate);  
  cd.setHeader("{ $fields..value }");  
  cd.display();  
  literal}  
  }  
  }/literal}  
</script>
```

You will now need to set the class variable `enable_in_hover` to true in the connector's constructor in `test/source/test.php`:

```
public function __construct(){  
  parent::__construct();  
  $this->has_testing_enabled = true;  
  $this->enable_in_hover = true;  
}
```

9) Provide optional mapping.php file.

This is another optional step where you may provide optional mapping entries for select modules in Sugar. In our `vardefs.php` file example in step 2, we enabled the hover link for the website field. To explicitly place this hover link on the website field for the Accounts module we provide the mapping entry as follows. A mapping.php file is needed though if you use the 'options' attribute for entries in the `vardefs.php` file.

```
<?php  
$mapping = array (  
  'beans' =>  
  array (  
-
```



```

'Accounts' =>
array (
'website' => 'website',
),
),
//options mapping
'options' =>
array (
'states_dom' =>
array(
'CA' => 'California',
'HI' => 'Hawaii',
),
),
);
?>

```

10) [Package your connector.](#)

The final step would be to zip your connector's contents along with a manifest.php file so that it may be installed by the Module Loader. Place the manifest.php file into folder that includes the "test" directory. Your zip file should have the following directory structure:

```

test/source/test.php
test/source/vardefs.php
test/source/config.php
test/source/mapping.php
-
test/language/en_us.lang.php
-
test/formatter/test.php
test/formatter/tpls/test.jpg
test/formatter/tpls/default.tpl
-

```

[../test/manifest.php](#) <--- in the folder that contains the "test" folder

A sample manifest.php file is as follows:

```

<?php
$manifest = array(
'acceptable_sugar_flavors' => array(
'CE',
'PRO',
'ENT',
),
'acceptable_sugar_versions' => array(
'5.2.0',
),
'is_uninstallable' => true,
'name' => 'Test Connector',
'description' => 'Connector for testing purposes only',
'author' => 'John Doe',
'published_date' => '2008/12/12',
'version' => '1.0',
'type' => 'module',
'icon' => "",
);

```



```

- $installdefs = array (
-   'id' => 'ext_rest_test',
-   'connectors' => array (
-     array (
-       'connector' => '<basepath>/test/source',
-       'formatter' => '<basepath>/test/formatter',
-       'name' => 'ext_rest_test',
-     ),
-   ),
- );
- ?>

```

Dynamic Teams

Dynamic Teams provides the ability to assign a record to multiple teams. This provides more flexibility in sharing data across functional groups.

Database Changes

The Sugar Professional and Sugar Enterprise versions create four new tables ([team_hierarchies](#), [team_sets](#), [team_sets_modules](#) and [team_sets_teams](#)). A description of each table is as follows:

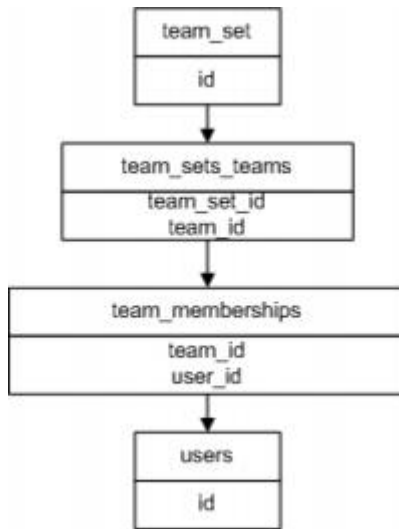
Table	Description
team_hierarchies	Not used currently, but added to provide future support for team hierarchies.
team_sets	Each record in this table represents a unique team combination. For example, each user's private team will have a corresponding team set entry in this table. A team set may also be comprised of one or more teams.
team_sets_teams	The team_sets_teams table maintains the relationships to determine which teams belong to a team set. Each table that previously used the team_id column to maintain team security now uses the team_set_id column's value to associate the record to team(s).
team_sets_modules	This table is used to manage team sets and keeps track of which modules have records that are or were associated to a particular team set.

In addition to the four new tables, each table that previously used the [team_id](#) column to store the team access information will now have a new [team_set_id](#) column to record the team set id value. This is also true in the case of upgrades from pre-5.5.x versions. However, custom modules created using the CE editions of the system that go through the flavor conversion to the PRO/ENT editions will not be upgraded to support teams. In such situations, it will still be the responsibility of the developer to provide the team security feature through Module Builder.



Team Security

The `team_sets_teams` table allows the system to check for permissions on multiple teams. The following diagram illustrates table relationships in SugarBean's `add_team_security_where_clause` method.



Using the `team_sets_teams` table the system will determine which teams are associated with the `team_set_id` and then look in the `team_memberships` table for users that belong to the team(s). On every Edit View screen, the user should be presented with a Teams widget which provides the ability to associate one or more teams to the record. The user can either perform a quick search to associate a team or they can select one or many teams from the popup. When the record is saved a `team_set_id` is associated to the record. A team set is simply a unique combination of teams. Sugar could have implemented Dynamic Teams as a many-many relationship in the database, but the idea of team sets takes advantage of combinations of teams that are re-used throughout the system. Rather than replicated sets of teams each time we create a record, we simply re-use the `team_set_id` thereby reducing the amount of duplicated data.

Team Sets

As mentioned above, Sugar implemented this feature not as a many-to-many relationship but as a one-to-many relationship. On each table that had a 'team_id' field we added a 'team_set_id' field. We have also added a 'team_sets' table which maintains the team_set_id, a 'team_sets_teams' table which relates a team set to the teams. When performing a team based query we use the 'team_set_id' field on the module table to join to 'team_sets_teams.team_set_id' and then join all of the teams associated with that set. Given the list of teams from `team_memberships` we can then decide if the user has access to the record.

Primary Team

The 'team_id' is still being used, not only to support backwards compatibility with workflow and reports but also to provide some additional features. When displaying a list we use the team set to determine whether the user has access to the record, but when displaying the data, we show the team from `team_id` in the list. When the user performs a mouse over on that team Sugar performs an Ajax call to display all of the teams associated with the record. This 'team_id' field is designated as the Primary Team because it is the first team shown in the list, and for sales territory management purposes, can designate the team that actually owns the record and can report on it.

[Adding Teams Programatically](#)



[To add teams to a record, you may first load the teams relationship on SugarBean and then pass in an Array of team ids to the add method.](#)

```
...  
$contact->load_relationship('teams');  
$contact->teams->add(array('East', 'West'));
```

...

[If there are already other teams assigned to this contact, the system will check to see if an entry in the team_sets table exists for the combination of teams specified. It will create entries in the team_sets and team_sets_teams tables if necessary to record the unique combination of teams. In addition, if you have a team_id value that is not in the list of teams in the set, then it will be added.](#)

[An example of the SOAP API to set teams is as follows](#)

```
...:  
//Create nusoapclient  
$this-> soapClient = new nusoapclient($soap_url);  
//Login  
$result = $this-> soapClient->call('login', ...);  
$session_id = $result['id'];  
$this-> soapClient->call('set_relationship',array('session'=>$this->  
> $session_id,'module_name'=>'Contacts', 'module_id'=>$contact_id, 'link_field_name' =>  
'teams', 'related_ids' => array('1', 'East', 'West')));
```

...

[It is important to note that when adding teams, additional teams that are not specified programmatically can be added. This will be the case if a person assigned to a record does not belong to any of the teams that will be associated with the record. In this scenario, we will add the assigned-to- user's private team into the list of teams. If you wish to disable this behavior, you can edit the system's config.php file and add the following entry:](#)

```
'disable_team_access_check' => true
```

Removing Teams Programmatically

[Removing teams is pretty simple. As before, you load the Teams relationship:](#)

```
$contact->load_relationship('teams');  
$contact->teams->remove(array('team1', 'team2'));
```

[One thing to note is that if one of the teams you are removing is the primary team as defined by the team_id column, then Sugar logs this and prevents the removal of this team. Because Sugar does not know what to do next if the primary team is removed, we prevent this action.](#)

Replacing Teams Programmatically

[As is often the case you may just want to replace all of the teams you have on a record so we have provided the replace method:](#)

```
$contact->load_relationship('teams');  
$contact->teams->replace(array('team1', 'team2'));
```

TeamSetLink

[You can define your own Link class. Typically any relationship in a class is handled by the data/Link.php class. As part of Dynamic Teams, we introduced the ability to provide your own custom Link class to handle some of the functionality related to managing relationships. The team_security parent vardefs in SugarObjects contains the following in the 'teams' field definition:](#)

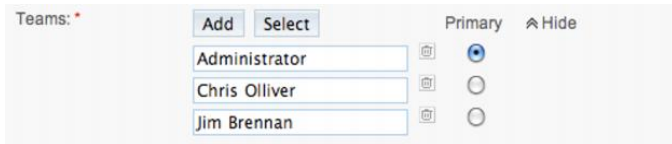
```
...  
'link_class' => 'TeamSetLink',  
'link_file' => 'modules/Teams/TeamSetLink.php',
```

[The link_class entry defines the class name we are using and the link_file tells us where that class file](#)



is located. This class should extend Link.php and you can then override some of the methods used to handle relationships such as 'add' and 'delete'.

The Dynamic Teams Widget



The only reason we present the widget here is to outline how each element in this widget is associated with an element in the system. This widget is rendered in place of the Teams relationship field. However, the old Teams relationship field will still work.

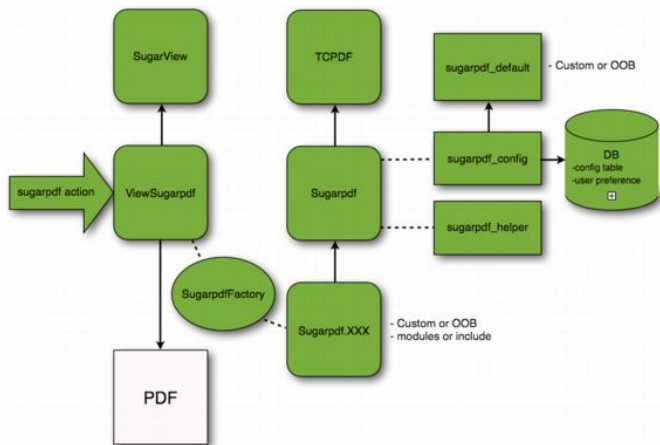
The Primary radio button will select the team which will be saved to the team_id field on the record. Every other field will be added as a team associated with the team set. Adding or removing teams here will result in this record being assigned a new team set. The team set may already exist, and if so that id is simply assigned to the record.

Printing to PDF

The OOB Quotes and Reports modules allow users to generate output in PDF format. Sugar uses the TCPDF engine in order to extend automatic PDF support for a much larger set of languages.

SugarPDF Architecture

The new PDF framework leverages the SugarCRM MVC architecture.



Key Classes

ViewSugarpdf (include/MVC/View/views/view.sugarpdf.php)

This new SugarView instance generates the PDF document. As with all SugarViews, ViewSugarpdf can be overridden with by

`modules/<myModule>/views/view.sugarpdf.php`

The Sugarpdf view can be launched by

`module=<mymodule>&action=sugarpdf&sugarpdf=<XXX>`



[TCPDF \(include/tcpdf/tcpdf.php\)](#)

[The TCPDF class is the original class from the TCPDF library. Modifications have been made to address some bugs that we encountered during our tests with this library.](#)

[Sugarpdf \(include/Sugarpdf/Sugarpdf.php\)](#)

[This class extends the TCPDF class. The following methods have been overridden in this class:](#)

- **[Header](#)** - This method override the regular Header() method to enable the custom image directory in addition to the OOB image directory.
- **[SetFont](#)** - This method override the regular SetFont() method to enable the custom font directory in addition to the OOB font directory.
- **[Cell](#)** - Handle HTML entity decode.
- **[getNumLines](#)** - This method is a fix for a better handling of the count. It handle the line break between words.

[Additional methods have been added to this class:](#)

- **[predisplay](#)** - preprocessing before the display method is called. Is intended to setup general PDF document properties like margin, footer, header, etc.
- **[display](#)** - performs the actual PDF content generation. This is where the logic to display output to the PDF should be placed.
- **[process](#)** - calls **`predisplay`** and **`display`**.
- **[writeCellTable](#)** - Method to print a table using the Cell print method of TCPDF
- **[writeHTMLTable](#)** - Method to print a table using the writeHTML print method of TCPDF

[Sugarpdf.XXX](#)

[Possible Locations :](#)

- [include/Sugarpdf/sugarpdf/sugarpdf.XXX.php](#)
- [modules/module_name/sugarpdf/sugarpdf.XXX.php](#)
- [custom/modules/module_name/sugarpdf/sugarpdf.XXX.php](#)



These classes extend the Sugarpdf class. They define a specific PDF view which is accessible with the following URL parameters:

- [module=module_name](#)
- [action=sugarpdf](#)
- [sugarpdf=XXX](#)

In this class the display method has to be redefined. It is also possible to override other methods like Header().

The process method of this class is a call from ViewSugarpdf.

The most relevant sugarpdf.XXX class is chosen by SugarpdfFactory.

SugarpdfFactory

The ViewSugarpdf class uses SugarpdfFactory to find the most relevant sugarpdf.XXX class which generates the PDF document for a given PDF view and module.

If one is not found, then Sugarpdf is used.

One of the following file will be used in the following order:

- 1) [custom/modules/my_module/sugarpdf/sugarpdf.XXX.php](#)
- 2) [modules/my_module/sugarpdf/sugarpdf.XXX.php](#)
- 3) [custom/include/Sugarpdf/sugarpdf/sugarpdf.XXX.php](#)
- 4) [include/Sugarpdf/sugarpdf/sugarpdf.XXX.php](#)
- 5) [include/Sugarpdf/sugarpdf.php](#)

SugarpdfHelper

This file is included by Sugarpdf.php. This php file is a utils file. It contains many functions that can be used to generate PDFs.

For example, to generate HTML code before using the writeHTML() method o TCPDFf

Available functions :

- [wrapTag, wrapTD, wrapTable ,etc. - These functions help to create an HTML code](#)
- [prepare_string - This function prepare a string to be ready for the PDF printing](#)
- [format_number_sugarpdf - This function is a copy of format_number\(\) from currency with a fix for sugarpdf](#)



FontManager

The FontManager class is a stand-alone class that manages all the fonts for TCPDF.

Functionality:

- [List all the available fonts from the OOB font directory and the custom font directory \(it can create a well formatted list for select tag\)](#)
- [Get the details of each listed font \(Type, size, encoding,...\) by reading the font php file](#)
- [Add a new font to the custom font directory from a font file and a metric file](#)
- [Delete a font from the custom font directory](#)

The font list build by the font manager with the [listFontFiles\(\)](#) or [getSelectFontList\(\)](#) is saved in a cached php file `cache/Sugarpdf/cachedFontList.php` to prevent to parse the fonts folder every time (if the cached file don't already exist). This file is automatically cleared when the `delete()` or `add()` methods are used.

[When you create a Font loadable module you will have to call the clearCachedFile\(\) method in a post_execute and post_uninstall actions to clear the cache. Like that, the cache will be rebuild with the last infos.](#)

Example of Font loadable module :

manifest.php

```
<?php
if(!defined('sugarEntry') || !sugarEntry) die('Not A Valid Entry Point');

-
$manifest = array(
-
'acceptable_sugar_versions' => array (
'regex_matches' => array (
0 => "5\.\5\.*",
),
),
'acceptable_sugar_flavors' => array (
0 => 'PRO',
1 => 'ENT',
),
'name' => 'Font MSung Light',
'description' => 'Font MSung Light (Trad. Chinese) for SugarPDF',
'author' => 'SugarCRM',
'published_date' => '2009-06-17',
'version' => '0.2',
'type' => 'module',
'icon' => '',
'is_uninstallable' => true,
);
global $installdefs;
$installdefs = array(
'id'=> 'Font MSung Light',
```



```
'copy' => array(
array('from'=> '<basepath>/new_files/custom/include/tcpdf/fonts/msungstdlight.php',
'to'=> 'custom/include/tcpdf/fonts/msungstdlight.php',
)
),
'pre_execute'=>array(
0 => '<basepath>/actions/pre_actions.php',
),
'pre_uninstall'=>array(
0 => '<basepath>/actions/pre_actions.php',
),
);
?>
```

[pre_actions.php](#)

```
<?php
require_once("include/Sugarpdf/FontManager.php");
$fontManager = new FontManager();
$fontManager->clearCachedFile();
?>
```

[Chain of Events](#)

1. [Call loadSugarpdf method of the SugarpdfFactory - use to determine which sugarpdf.XXX class to load depending of the sugarpdf URL parameter.](#)
2. [Call the process method of the determined sugarpdf.XXX class - this method builds/ prints the PDF](#)
3. [Call the output method of the determined sugarpdf.XXX class - this method outputs the PDF to the user](#)

[PDF settings \(user and system\)](#)

[sugarpdf_config.php](#)

[The first existing file from the following is used for TCPDF class configuration:](#)

1. [custom/include/Sugarpdf/sugarpdf_config.php](#)
2. [include/Sugarpdf/sugarpdf_config.php](#)

[The properties set in this file will affect all the generated SugarPDF files.](#)

[Check the comments in the file for more details.](#)

[The major settings configured in this file are used in the preDisplay method of the Sugarpdf class.](#)

[sugarpdf_default.php](#)

[include/Sugarpdf/sugarpdf_default.php](#) is used to store the default value of the PDF settings (system and user).

[You can overwrite any default value using custom/include/Sugarpdf/sugarpdf_default.php.](#)

[For example if you want to set the default right margin to 25 pdf units, create or modify the file](#)

[custom/include/Sugarpdf/sugarpdf_default.php](#) with this content:

[\\$sugarpdf_default\['PDF_MARGIN_LEFT'\]=25;](#)



Mechanism

[sugarpdf_config](#) sets all PDF settings in constant variables.

[The values used to set the variables come from different sources in a specific order:](#)

1. [DB](#)
 - o [Config table for the system settings \(category : sugarpdf\)](#)
 - o [User_preferences table for the user settings](#)
2. [Default value from sugarpdf_default.php in custom directory](#)
 - o [custom/include/Sugarpdf/sugarpdf_default.php](#)
3. [Default value from sugarpdf_default.php in OOB directory](#)
 - o [include/Sugarpdf/sugarpdf_default.php](#)

[If you modify the settings from the user interface \(My Account screen, or PDF settings in admin\), the new values will be saved in the DB and will override the default value from sugarpdf_default.php .](#)

[The Restore button in PDF settings deletes all sugarpdf settings saved in the config table.](#)

The custom directory

Fonts

[location : custom/include/tcpdf/fonts/](#)

[The font manager](#) parses this directory for available custom font files in addition to the OOB directory.

Logos

[location : custom/themes/default/images](#)

[All the uploaded logos from PDF settings](#) are copied to this location.

sugarpdf_default.php

[location : custom/include/Sugarpdf/sugarpdf_default.php](#)

[This file is included after the OOB sugarpdf_default.php file. It can be used to overwrite any default value for the PDF settings.](#)

sugarpdf.XXX.php

[location : custom/include/Sugarpdf/sugarpdf/sugarpdf.XXX.php](#)

OR

[location : custom/modules/<AnyModule>/sugarpdf/sugarpdf.XXX.php](#)

[These files are used to override a sugarpdf.XXX.php class for a specific module or for the whole application.](#)

Adding New PDF Templates

[To create a new SugarPDF document, check existing examples in Quotes, Reports, and Projects first.](#)

Steps

[\(For OOB remove "custom/"\)](#)



1. [Add the file](#) `custom/modules/<myModule>/sugarpdf/sugarpdf.<mySugarpdf>.php`
2. [Into the created file](#) :
 - a) [require include/Sugarpdf/Sugarpdf.php](#)
 - b) [create the class <Mymodule>Sugarpdf<mySugarpdf> which extend Sugarpdf](#)
 - c) [Override display\(\) and preDisplay\(\) methods.](#)
3. [Create a button in the UI to access SugarPDF \(module=<Mymodule>&action=sugarpdf&sugarpdf=<mySugarpdf>\).](#)

```

<?php
if(!defined('sugarEntry') || !sugarEntry) die('Not A Valid Entry Point');
require_once('include/Sugarpdf/Sugarpdf.php');
class <Mymodule>Sugarpdf<mySugarpdf> extends Sugarpdf{
function preDisplay(){
parent::preDisplay();
//...
}
function display(){
//Create new page
$this->AddPage();
$this->SetFont(PDF_FONT_NAME_MAIN,"PDF_FONT_SIZE_MAIN);
$this->fileName = "test.pdf";
$this->Ln1();
$this->writeHTML("<p>Bonjour, comment allez vous?</p>");
//...
}
// ... other method that you would like to override
}

```

[The useful methods to create a SugarPDF document are :](#)

- [Multicell\(\)](#)
- [writeHTML\(\)](#)
- [writeCellTable\(\)](#)
- [writeHTMLTable\(\)](#)



Smarty

You can choose to create a PDF from a Smarty template. You will have to assign variables to the Smarty template, generate the HTML with Smarty and pass it to TCPDF using the `WriteHTML()` method. The **SugarpdfSmarty** class is an helper class which support part of these steps.

Warning : HTML and CSS support in `writeHTML()` is limited. For more details, read the TCPDF documentation

[Example in the Contacts module :](#)

- [Add `custom/modules/Contacts/sugarpdf/sugarpdf.test.php`](#) with this content :

```
<?php
if(!defined('sugarEntry') || !sugarEntry) die('Not A Valid Entry Point');

require_once('include/Sugarpdf/sugarpdf/sugarpdf.smarty.php');

/**
 * This is an helper class to generate PDF using smarty template.
 * You have to extend this class, set the templateLocation and assign
 * the Smarty variables in the preDisplay method.
 * @author bsoufflet
 */
class ContactsSugarpdfTest extends SugarpdfSmarty{
function preDisplay(){
parent::preDisplay();
$this->templateLocation = "custom/modules/Contacts/tpls/test.tpl";
$this->ss->assign("AA",$this->bean->last_name);
$this->ss->assign("BB","2");
$this->ss->assign("CC","3");
$this->ss->assign("DD","4");
}
}
```

- [Add `custom/modules/Contacts/tpls/test.tpl`](#) with this content :

```
<p>This is just an example of html code to demonstrate some supported CSS inline styles.</p>
<div style="font-weight: bold;">{$AA}</div>
<div style="text-decoration: line-through;">{$BB}</div>
<div style="text-decoration: underline line-through;">{$CC}</div>
<div style="color: rgb(0, 128, 64);">{$DD}</div>
<div style="background-color: rgb(255, 0, 0); color: rgb(255, 255, 255);">background
color</div>
<div style="font-weight: bold;">bold</div>
<div style="font-size: xx-small;">xx-small</div>
<div style="font-size: small;">small</div>
<div style="font-size: medium;">medium</div>
<div style="font-size: large;">large</div>
<table>
<tr><td>a</td><td>b</td></tr>
<tr><td>c</td><td>d</td></tr>
</table>
```



- [Generate the PDF file using this URL : `module=Contacts&action=sugarpdf&sugarpdf=test`](#)

[How to Add a New Font \(without Font Manager\)](#)

[If you have a TCPDF font file \(.php\) \(and the .z and .ctg.z files if they are needed\):](#)

- [Create the directory `custom/include/tcpdf/fonts` if it is not created](#)
- [Copy these files into the directory `custom/include/tcpdf/fonts`](#)

[If you have font file \(.ttf or .otf\):](#)

- [Follow steps 1 to 5 of the \[TCPDF Fonts webpage\]](#)
- [Create the directory `custom/include/tcpdf/fonts` if it is not created](#)
- [Copy all the generated files \(.php AND .z and .ctg.z if generated\) into the directory `custom/include/tcpdf/fonts`](#)

[makefont.php and the utilities use in the TCPDF Fonts webpage can be found in the TCPDF package in `fonts/utls`. You can download the TCPDF package \[\\[here\\]\]\(#\).](#)

[How to Add More Configurations to PDF Settings](#)

- [Create `custom/modules/Configurator/metadata/SugarpdfSettingsdefs.php`](#)
- [Add any settings that you would like to add to the "basic" or "logo" section \("Advanced" section is not available without modifying an OOB file\)](#)

[Example :](#)

```
require_once('include/Sugarpdf/sugarpdf_config.php');
\SugarpdfSettings["sugarpdf_pdf_creator"]=array(
    "label"=>$mod_strings["PDF_CREATOR"],
    "info_label"=>$mod_strings["PDF_CREATOR_INFO"],
    "value"=>PDF_CREATOR,
    "class"=>"basic",
    "type"=>"text",
    "required"=>"true"
);
```

[You can also completely overwrite the SugarpdfSettings array by writing this for example in the custom SugarpdfSettingsdefs.php file:](#)

```
require_once('include/Sugarpdf/sugarpdf_config.php');

\SugarpdfSettings = array(
    "sugarpdf_pdf_title"=>array(
        "label"=>$mod_strings["PDF_TITLE"],
        "info_label"=>$mod_strings["PDF_TITLE_INFO"],
        "value"=>PDF_TITLE,
        "class"=>"basic",
```



```
"type"=>"text",
),
"sugarpdf_pdf_subject"=>array(
"label"=>$mod_strings["PDF_SUBJECT"],
"info_label"=>$mod_strings["PDF_SUBJECT_INFO"],
"value"=>PDF_SUBJECT,
"class"=>"basic",
"type"=>"text",
)
);
```

[In this example, only two fields will be available in PDF settings.](#)

Note : You can view all the available settings in [modules/Configurator/metadata/SugarpdfSettingsdefs.php](#) (commented and not commented).

[How to Create a Portal API User](#)

[If you are using Sugar Professional, and would like to build a custom portal using the Portal API, you will need to create a Portal API User to communicate with the Sugar server.](#)

[You can enable the functionality as follows:](#)

1. [Add the following parameter to `config_override.php`:](#)

```
$sugar_config['enable_web_services_user_creation']= true
```

-

2. [Log into Sugar as the administrator and navigate to the Administration Home page.](#)
3. [Click "User Management".](#)
4. [To register a new user, from the Users tab, select "Create Portal API User".](#)

[You can now pass this user to the Portal API connect calls.](#)

[How to Enable Unsupported Email Configurations](#)

[To improve performance, the following email settings have been disabled, by default:](#)

- [The option to select Sendmail for outbound emails.](#)

[While you can enable these settings, support from SugarCRM for these features is no longer available. To re-enable these settings, you will need to add a configuration parameter to the `config_override.php` file with the value set to true.](#)

[The following table outlines the parameter key that must be set for the desired result:](#)

-
[For example, to enable the POP3 protocol, the `config_override.php` file would have the following entry set:](#)

```
$sugar_config['allow_pop_inbound']= true;
```





Sugar Logic

1. [Overview](#)
2. [Terminology](#)
3. [Sugar Formula Engine](#)
 - 3.1. [Formulas](#)
 - 3.2. [Types](#)
 - 3.2.1. [Number Type](#)
 - 3.2.2. [String Type](#)
 - 3.2.3. [Boolean Type](#)
 - 3.2.4. [Enum Type \(list\)](#)
 - 3.3. [Functions](#)
 - 3.4. [Triggers](#)
 - 3.5. [Actions](#)
 - 3.6. [Dependencies](#)
 - 3.6.1. [Dependent Fields](#)
 - 3.6.2. [Dependent Dropdown](#)
4. [Using Sugar Logic Directly](#)
 - 4.1. [Creating a Custom Dependency for a View](#)
 - 4.2. [Using Dependencies in Logic Hooks](#)
5. [Extending Sugar Logic](#)
6. [Writing a Custom Action](#)
7. [Accessing an External API with a Sugar Logic Action](#)
8. [Updating the Cache](#)

Overview

Sugar Logic, a new feature in Sugar Enterprise and Sugar Professional, is designed to allow custom business logic that is easy to create, manage, and reuse on both the server and client.

Sugar Logic is made up of multiple components which build off each other and is extensible at every step. The base component is the Sugar Formula Engine which parses and evaluates human readable formulas. Dependencies are units made up of triggers and actions that can express custom business logic. Each dependency defines a list of actions to be performed depending on the outcome of a trigger formula.



Terminology

- *Formula*: An expression that conforms to the Formula Engine syntax consisting of nested **functions** and **field variables**.
- *Function*: A method which can be called in a **formula**.
- *Trigger*: A **Formula** which evaluates to either true or false. Triggers are evaluated whenever a field in the equation is updated or when a record is retrieved/saved.
- *Action*: A method which modifies the current record or layout in some way.
- *Dependency*: A complete logical unit which includes a **trigger** and one or more **actions**.

Sugar Formula Engine

Formulas

The fundamental object is called a Formula. A Formula can be evaluated for a given record using the Sugar Logic parser.

Some example formulas are:

Basic addition: `add(1, 2)`

Boolean values: `not(equal($billing_state, "CA"))`

Calculation: `multiply(number($employees), $seat_cost, 0.0833)`

Types

Sugar Logic has several fundamental types. They are: *number*, *string*, *boolean*, and *enum (lists)*. Functions may take in any of these type or combinations thereof and return as output one of these types. Fields may also often have their value set to only a certain type.

Number Type

Number types essentially represent any real number (which includes positive, negative, and decimal numbers). They can be plainly typed in as input to any function. For example, the operation (10 + 10 + (15 - 5)) can be performed as follows:

```
add(10, 10, subtract(15, 5))
```

String Type

A string type is very much like a string in most programming languages. However, it can only be declared within *double quotes*. For example, consider this function which returns the length of the input string:

```
strlen("Hello World")
```



The function would appropriately return the value 11.

Boolean Type

A boolean type is simple. It can be one of two values: **true** or **false**. This type mainly acts as a flag, as in whether a condition is met or not. For example, the function **contains** takes in as input two strings and returns **true** if the first string contains the second string or **false** otherwise.

```
and(contains("Hello World", "llo Wo"), true)
```

The function would appropriately return the value **true**.

Enum Type (list)

An enum type is a collection of items. The items need to all be of the same type, they can be varied. An enum can be declared using the **enum** function as follows:

```
enum("hello world!", false, add(10, 15))
```

Alternatively, the **createList** function (an alias to **enum**) can also be used to create enums in a formula.

```
createList("hello world!", false, add(10, 15))
```

This function would appropriately return an enum type containing *"hello world!", false, and 25* as its elements.

Functions

Functions are methods to be used in formulas. Each function has a function name, a parameter count, a parameter type requirement, and returns a value. Some functions such as **add** can take any number of parameters. For example: **add(1)**, **add(1, 2)**, **add(1, 2, 3)** are all valid formulas. Functions are designed to produce the same result whether executed on the server or client.

Triggers

A Trigger is an object that listens for changes in field values and after a change is performed, triggers the associated Actions in a Dependency.

Actions

Actions are functions which modify a target in some way. Most Actions require at least two parameters: a target and a formula. For example, a style action will change the style of a field based on a passed in string formula. A value action will update a value of a field by evaluating a passed in formula.

Dependencies

A Dependency describes a set of rules based on a trigger and a set of actions. Examples include a field whose properties must be updated dynamically or a panel which must be hidden when a drop down value is not selected. When a Dependency is *triggered* it will appropriately carry out the action it is designed to. A basic Dependency is when a field's value is dependent on the result of evaluating a Expression. For example, consider a page with five fields with It's "a", "b", "c", "d", and "sum". A generic Dependency can be created between "sum" and the other four fields by using an Expression that links them together, in this case an *add* Expression. So we can define the Expression in this manner:



'**add(\$a, \$b, \$c, \$d)**' where each field id is prefixed with a dollar (\$) sign so that the value of the field is dynamically replaced at the time of the execution of the Expression.

An example of a more customized Dependency is when the field's style must be somehow updated to a certain value. For example, the DIV with id "temp" must be colored blue. In this case we need to change the *background-color* property of "temp". So we define a StyleAction in this case and pass it the field id and the style change that needs to be performed and when the StyleAction is triggered, it will change the style of the object as we have specified.

Sugar Logic Based Features

Calculated Fields

Fields with calculated values can now be created from within Studio and Module Builder. The values are calculated based on Sugar Logic formulas. These formulas are used to create a new dependency that are executed on the client side in edit views and the server side on save. The formulas are saved in the *vardef* or *vardef* extensions and can be created and edited directly. For example, the metadata for a simple calculated commission field in opportunities might look like:

```
'commission_c' => array(  
  'name' => 'commission_c',  
  'type' => 'currency',  
  'calculated' => true,  
  'formula' => 'multiply($amount, 0.1)',  
  //enforced causes the field to appear read-only on the layout  
  'enforced' => true  
)
```

Dependent Fields

A dependent field will only appear on a layout when the associated formula evaluates to the boolean value "true". Currently these cannot be created through Studio and must be enabled manually with a custom *vardef* or *vardef* extension. The "dependency" properties contains the expression that defines when this field should be visible. An example field that only appears when an account has an annual revenue greater than one million.

```
'dep_field' => array(  
  'name' => 'dep_field',  
  'type' => 'varchar',  
  'dependency' => 'greaterThan($annual_revenue, 1000000)',  
)
```

Dependent Dropdown

Dependent Dropdowns are dropdowns who's options change when the selected value in a trigger dropdown changes. The metadata is defined in the *vardefs* and contains two major components, a "trigger" id which is the name of the trigger dropdown field and a 'visibility grid' that defines the set of options available for each key in the trigger dropdown. For example, you could define a sub-industry field in accounts whose available values depend on the industry field.

```
'sub_industry_c' => array(  
  'name' => 'sub_industry_c',  
  'type' => 'enum',  
  'options' => 'sub_industry_dom',  
  'visibility_grid' => array(  
    'trigger' => 'industry',  
    'values' => array(  
      'Education' => array('primary', 'secondary', 'college'),  
      'Engineering' => array('mechanical', 'electrical', 'software'),  
    ),  
  ),  
)
```



)

Using Sugar Logic Directly

Creating a Custom Dependency for a View

Dependencies can also be created and executed outside of the built in features. For example, if you wanted to have the description field of the Calls module become required when the subject contains a specific value, you could extend the calls edit view to include that dependency.

```
<?php
//custom/modules/Calls/views/view.edit.php

require_once('include/MVC/View/views/view.edit.php');
require_once("include/Expressions/Dependency.php");
require_once("include/Expressions/Trigger.php");
require_once("include/Expressions/Expression/Parser/Parser.php");
require_once("include/Expressions/Actions/ActionFactory.php");
class CallsViewEdit extends ViewEdit {
function CallsViewEdit(){
parent::ViewEdit();
}
function display() {
parent::display();
$dep = new Dependency("description_required_dep");
$triggerExp = 'contains($name, "important");
//will be array('name')
$triggerFields = Parser::getFieldsFromExpression($triggerExp);
$dep->setTrigger(new Trigger($triggerExp, $triggerFields));
//Set the description field to be required if "important" is in the call subject
$dep->addAction(ActionFactory::getNewAction('SetRequired', array(
'target' => 'description',
'value' => 'true')
));
//Set the description field to NOT be required if "important" is NOT in the call subject
$dep->addFalseAction(ActionFactory::getNewAction('SetRequired', array(
'target' => 'description',
'value' => 'false')
));
//Evaluate the trigger immediatly when the page loads
$dep->setFireOnLoad(true);
$javascript = $dep->getJavascript();
echo <<<EOQ
<script type=text/javascript>
SUGAR.forms.AssignmentHandler.registerView('EditView');
{$javascript}
</script>
EOQ;
}
}
```

The above code creates a new Dependency object with a trigger based on the 'name' (Subject) field in of the Calls module. It then adds two actions. The first will set the description field to be required when the trigger formula evaluates to true (when the subject contains "important"). The second will fire when the trigger is false and removes the required property on the description field. Finally, the javascript version of the Dependency is generated and echoed onto the page.



Using Dependencies in Logic Hooks

Dependencies can not only be executed on the server side, but can be useful entirely on the server. For example, you could have a dependency that sets a rating based on a formula defined in a language file.

```
require_once("include/Expressions/Dependency.php");
require_once("include/Expressions/Trigger.php");
require_once("include/Expressions/Expression/Parser/Parser.php");
require_once("include/Expressions/Actions/ActionFactory.php");
```

```
class Update_Account_Hook {
function updateAccount($bean, $event, $args) {
$formula = translate('RATING_FORMULA', 'Accounts');
$triggerFields = Parser::getFieldsFromExpression($formula);
$dep = new Dependency('updateRating');
$dep->setTrigger(new Trigger('true', $triggerFields));
$dep->addAction(ActionFactory::getNewAction('SetValue',
array('target' => 'rating', 'value' => $formula)));
$dep->fire($bean);
}
}
```

Extending Sugar Logic

The most important feature of Sugar Logic is that it is simply and easily extendable. Both custom formula functions and custom actions can be added in an upgrade safe manner to allow almost any custom logic to be added to Sugar.

Writing a Custom Formula Function

Custom functions will be stored in "custom/include/Expressions/Expression/{**Type**}/{**Function_Name**}.php". The first step in writing a custom function is to decide what category the function falls under. Take for example a function for calculating the **factorial** of a number. In this case we will be returning a number so we will create a file in "custom/include/Expressions/Expression/**Numeric**/" called "FactorialExpression.php". In the new PHP file we just created, we will define a class called **FactorialExpression** that will extend **NumericExpression**. All formula functions must follow the format "{functionName}Expression.php" and the class name must match the file name. Next we need to decide what parameters the function will accept. In this case, we need take in a single parameter, the number to return the factorial of. Since this class will be a sub-class of *NumericExpression*, it by default accepts only *numeric* types, we need not worry about specifying the type requirement.

Next, we must define the logic behind evaluating this expression. So we must override the abstract **evaluate()** function. The parameters can be accessed by calling an internal function **getParameters()** which returns the parameters passed in to this object. So with all this information we can go ahead and write the code for the function.

```
<?php
require_once('include/Expressions/Expression/Numeric/NumericExpression.php');
class FactorialExpression extends NumericExpression {
function evaluate() {
$params = $this->getParameters();
// params is an Expression object, so evaluate it
// to get its numerical value
$number = $params->evaluate();
// exception handling
if ( ! is_int( $number ) ) {
throw new Exception("factorial: Only accepts integers");
}
}
```



```

if ( $number < 0 ) {
throw new Exception("factorial: The number must be positive");
}

// special case 0! = 1
if ( $number == 0 ) return 1;

// calculate the factorial
$factorial = 1;

for ( $i = 2 ; $i <= $number ; $i ++ )
$factorial = $factorial * $i;

return $factorial;
}
// Define the javascript version of the function
static function getJSEvaluate() {
return <<<EOQ
var params = this.getParameters();
var number = params.evaluate();
// reg-exp integer test
if ( !/^\\d*$/.test(number) )
throw "factorial: Only accepts integers";
if ( number < 0 )
throw "factorial: The number must be positive";

// special case, 0! = 1
if ( number == 0 )
return 1;

// compute factorial
var factorial = 1;
for ( var i = 2 ; i <= number ; i ++ )
factorial = factorial * i;
return factorial;
EOQ;
}

function getParameterCount() {
return 1; // we only accept a single parameter
}

static function getOperationName() {
return "factorial"; // our function can be called by 'factorial'
}
}

```

One of the key features of Sugar Logic is that the functions should be defined in both php and javascript, and have the same functionality under both circumstances. As you can see above, the **getJSEvaluate()** method should return the JavaScript equivalent of your **evaluate()** method. The JavaScript code is compiled and assembled for you after you run the "Rebuild Sugar Logic Functions" script through the admin panel.

Writing a Custom Action

Using custom actions, you can easily create reusable custom logic or integrations that can include user-editable logic using the Sugar Formula Engine. Custom actions will be stored in "custom/include/Expressions/Actions/{**ActionName**}.php". Actions files must end in "Action.php" and the class defined



in the file must match the file name and extend the "AbstractAction" class. The basic functions that must be defined are "fire", "getDefinition", "getActionName", "getJavascriptClass", and "getJavascriptFire". Unlike functions, there is no requirement that an action works exactly the same both server and client side as this is not always sensible or feasible.

A simple action could be a "**WarningAction**" that shows an alert warning the user that something may be wrong, and logs a message to the **sugarcrm.log** file if triggered on the server side. It will take in a message as a formula so that the message can be customized at run time. We would do this by creating a php file in "custom/include/Expressions/Actions/**WarningAction.php**". containing the following code:

```
<?php
require_once("include/Expressions/Actions/AbstractAction.php");
class WarningAction extends AbstractAction{
protected $messageExp = "";
function SetZipCodeAction($params) {
$this->messageExp = $params['message'];
}
/**
 * Returns the javascript class equivalent to this php class
 * @returnstring javascript.
 */
static function getJavascriptClass() {
return "
SUGAR.forms.WarningAction = function(message) {
this.messageExp = message;
};
//Use the sugar extend function to extend AbstractAction
SUGAR.util.extend(SUGAR.forms.WarningAction, SUGAR.forms.AbstractAction, {
//javascript execution code
exec : function()
{
//assume the message is a formula
var msg = SUGAR.forms.evalVariableExpression(this.messageExp);
alert(msg.evaluate());
}
});";
}
/**
 * Returns the javascript code to generate this actions equivalent.
 * @returnstring javascript.
 */
function getJavascriptFire() {
return "new SUGAR.forms.WarningAction('{ $this->messageExp}')";
}
/**
 * Applies the Action to the target.
 * @paramSugarBean $target
 */
function fire(&$target) {
//Parse the message formula and log it to fatal.
$expr = Parser::replaceVariables($this->messageExp, $target);
$result = Parser::evaluate($expr)->evaluate();
$GLOBALS['log']->warn($result);
}
/**
 * Returns the definition of this action in array format.
 */
function getDefinition() {
return array(
"message" => $this->messageExp,
```



```

);
}
/**
 * Returns the short name used when defining dependencies that use this action.
 */
static function getActionName() {
return "Warn";
}
}

```

Accessing an External API with a Sugar Logic Action

Let us say we were building a new Action called "**SetZipCodeAction**" that uses the yahoo geocode API to get the zip code for a given street + city + state address.

Since the Yahoo Geocode API requires JSON requests and returns XML data, we will have to write both php and javascript code to make and interpret the requests. Because accessing external APIs in a browser is considered cross site scripting, a local proxy will have to be used. We will also allow the street, city, state parameters to be passed in as formulas so the action could be used in more complex **Dependencies**.

First, we should add a new action that acts as the proxy for retrieving data from the Yahoo API. The easiest place to add that would be a custom action in the "Home" module. The file that will act as the proxy will be "custom/modules/Home/geocode.php". It will take in the parameters via a REST call, make the call to the Yahoo API, and return the result in JSON format.

geocode.php contents:

```

<?php
function getZipCode($street, $city, $state) {
$appID = "6ruuUKjV34Fydi4TE.ca.I02rWh.9LTMPqQnSQo4QsCnjF5wIvyYRSXPIzqIDbI.jfE-";
$street = urlencode($street);
$city = urlencode($city);
$state = urlencode($state);
$base_url = "http://local.yahooapis.com/MapsService/V1/geocode?";
$params = "appid={$appID}&street={$street}&city={$city}&state={$state}";
//use file_get_contents to easily make the request
$response = file_get_contents($base_url . $params);
//The PHP XML parser is going to be overkill in this case, so just pull the zipcode with a regex.
preg_match('/\<Zip\>([\d-]*)\</Zip\>/', $response, $matches);
return $matches[1];
}

if (!empty($_REQUEST['execute'])) {
if (empty($_REQUEST['street']) || empty($_REQUEST['city']) || empty($_REQUEST['state']))
echo("Bad Request");
else
echo json_encode(array('zip' => getZipCode($_REQUEST['street'], $_REQUEST['city'], $_REQUEST['state'])));
}

```

Next we will need to map the geocode action to the geocode.php file. This is done by adding an action map to the Home Module. We need to create the file "custom/modules/Home/action_file_map.php" and add the following line of code:

```

<?php
$action_file_map['geocode'] = 'custom/modules/Home/geocode.php';

```

Finally we are ready to write our Action! We will start by creating the file "custom/include/Expressions/Actions/**SetZipCodeAction.php**". It will use the proxy function directly from the php side and make an asynchronous call on the javascript side to the proxy.

SetZipCodeAction.php contents:



```

<?php
require_once("include/Expressions/Actions/AbstractAction.php");

class SetZipCodeAction extends AbstractAction{
protected $target = "";
protected $streetExp = "";
protected $cityExp = "";
protected $stateExp = "";
function SetZipCodeAction($params) {
$this->target = empty($params['target']) ? " " : $params['target'];
$this->streetExp = empty($params['street']) ? " " : $params['street'];
$this->cityExp = empty($params['city']) ? " " : $params['city'];
$this->stateExp = empty($params['state']) ? " " : $params['state'];
}
static function getJavascriptClass() {
return "
SUGAR.forms.SetZipCodeAction = function(target, streetExp, cityExp, stateExp) {
this.street = streetExp;
this.city = cityExp;
this.state = stateExp;
this.target = target;
};
SUGAR.util.extend(SUGAR.forms.SetZipCodeAction, SUGAR.forms.AbstractAction, {
targetUrl:'index.php?module=Home&action=geocode&to_pdf=1&execute=1&',
exec : function()
{
var street = SUGAR.forms.evalVariableExpression(this.street).evaluate();
var city = SUGAR.forms.evalVariableExpression(this.city).evaluate();
var state = SUGAR.forms.evalVariableExpression(this.state).evaluate();
var params = SUGAR.util.paramsToUrl({
street: encodeURIComponent(street),
city: encodeURIComponent(city),
state: encodeURIComponent(state)
});
YAHOO.util.Connect.asyncRequest('GET', this.targetUrl + params, {
success:function(o){
var resp = YAHOO.lang.JSON.parse(o.responseText);
SUGAR.forms.AssignmentHandler.assign(this.target, resp.zip);
},
scope:this
});
}
});";
}

function getJavascriptFire() {
return "new SUGAR.forms.SetZipCodeAction('{ $this->target}', '{ $this->streetExp}', "
. "'{ $this->cityExp}', '{ $this->stateExp}'");
}
function fire(&$bean) {
require_once("custom/modules/Home/geocode.php");
$vars = array('street' => 'streetExp', 'city' => 'cityExp', 'state' => 'stateExp');
foreach($vars as $var => $exp) {
$toEval = Parser::replaceVariables($this->$exp, $bean);
$var = Parser::evaluate($toEval)->evaluate();
}
$target = $this->target;
$bean->$target = getZipCode($street, $city, $state);
}

```



```
}  
function getDefinition() {  
return array(  
"action" => $this->getActionName(),  
"target" => $this->target,  
);  
}  
static function getActionName() {  
return "SetZipCode";  
}  
}
```

Once you have the action written, you need to call it somewhere in the code. Currently this must be done as shown above using either custom views, logic hooks, or custom modules. This will change in the future and creating custom dependencies and taking advantage of custom actions should become a simpler process requiring little to no custom code.

Updating the Cache

The `updatecache.php` script in the main directory traverses the Expression directory for every file that ends with "Expression.php" (ignoring a small list of excepted files) and constructs both a PHP and a JavaScript functions cache file which resides in "cache/Expressions/functions_cache.js" and "cache/Expressions/functionmap.php". So after you create your custom functions, you should run this script to integrate it into the entire framework. This can be done using the "Rebuild Sugar Logic Functions" link on the Admin Repair page.

Copyright 2004-2010 SugarCRM Inc.
[Community Edition License](#)

