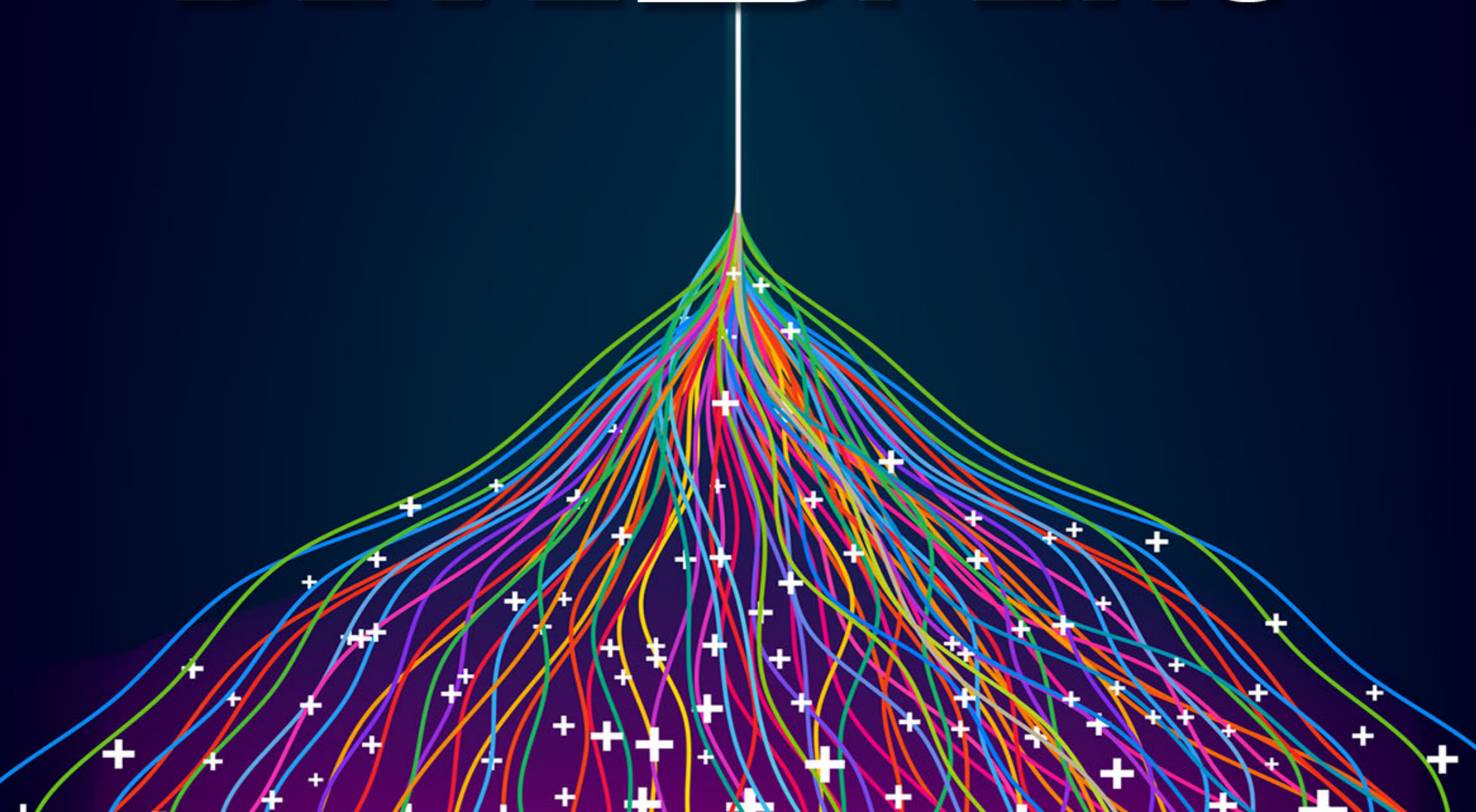
Jim Mackin

SuiteCRM for DEVELOPERS



SuiteCRM for Developers

Getting started with developing for SuiteCRM

Jim Mackin

This book is for sale at http://leanpub.com/suitecrmfordevelopers

This version was published on 2015-05-22

Tweet This Book!

Please help Jim Mackin by spreading the word about this book on Twitter!

The suggested hashtag for this book is #SuiteCRMForDevelopers.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#SuiteCRMForDevelopers

Contents

1. Introduction

What is SuiteCRM

The story of SuiteCRM[1] starts with SugarCRM. SugarCRM was founded in 2004 and consisted of an open source version (called Community Edition) and various paid for versions. However trouble started brewing when it appeared that SugarCRM would not be releasing a Community Edition of SugarCRM 7 and would be providing limited, if any, updates to the Community Edition.

Enter SuiteCRM. SalesAgility forked Community Edition to create SuiteCRM and also added various open source plugins to add improved functionality.

This book

This book is intended for developers who are familiar (or at least acquainted) with using SuiteCRM but want to perform their own customisations. SuiteCRM is a large and mature piece of software so it is impractical for a book to cover all aspects of the software. I've tried to add the most important parts which should allow you to make the changes you need in 99% of situations. There is a further resources chapter at the end of this book to help out in those 1% of cases. With that being said if you feel there is anything important I have left out (or worse, anything incorrect in the book) please let me know. I can be contacted at JSMackin.co.uk[2].

Reading this book

Each chapter in this book is intended to be self contained so the reader can jump to interesting chapters. Where there is some overlap this is usually indicated with links to the relevant chapters.

Some parts of this book may refer to file paths or other parts of code that can have a variable value, for example controller names contain the module name or a file with an arbitrary name. In this case these will be marked in the form `<TheModuleName>`, `<TheFileName>` or something else suitable. In these cases you can substitute something appropriate (such as `Accounts` or `MyNewFile`).

Setting up SuiteCRM

In this book we'll be using SuiteCRM v7.1.5 which is the latest at time of writing. For up to date versions of the installation instructions see the SuiteCRM wiki at suitecrm.com/wiki/index.php/Installation[3].

[1]https://www.suitecrm.com
[2]http://www.jsmackin.co.uk
[3]https://suitecrm.com/wiki/index.php/Installation

Website

The SuiteCRM installer can be found at SuiteCRM.com[4]. I would recommend SuiteCRM MAX as I prefer to start with a full interface and customise it as needed.

GitHub

SuiteCRM is also available on GitHub[5] at github.com/salesagility/SuiteCRM[6]. Each SuiteCRM version is tagged so you can easily grab the version you need.

Initial Tweaks

After the initial install there are a few tweaks you may want to make on an instance you are developing on. These changes should improve your development flow and productivity as well as help identify issues if they occur.

Developer Mode

SuiteCRM will cache various files that it processes, such as Smarty templates. Developer mode will turn off some of the caching so that changes to files will be seen immediately (though this isn't always the case - as is the case with extensions). This can be enabled either through the config file or via the General settings page inside admin.

Log Level

The default log level of SuiteCRM is `fatal`. This is a good default for production instances but you may want to increase the log level to `info` or `debug`. This will make the log output more verbose so, should anything go wrong, you'll have something to refer to. See the chapter on logging for more information.

Display errors

You'll also want to turn off display errors. Unfortunately at the moment SuiteCRM has various notices and warnings out of the box. With `display_errors` on this can sometimes cause AJAX pages and the link to break.

With this being said you should be checking the PHP error logs or selectively enabling `display_errors` to ensure that the code you are creating is not creating additional notices, warnings or errors.

[4]https://suitecrm.com/
[5]http://github.com
[6]https://github.com/salesagility/SuiteCRM

XDebug

XDebug[7] is a PHP extension which provides profiling and debugging capabilities to PHP. This can massively improve developer productivity by simplifying development and, particularly, tracking down any issues. See the XDebug site for information on XDebug.

[7] http://xdebug.org

2. SuiteCRM Directory Structure

`cache`
: Contains cache files used by SuiteCRM including compiled smarty templates, grouped vardefs, minified and grouped JavaScript. Some modules and custom modules may also store (temporary) module specific info here.

`custom`
: Contains user and developer customisations to SuiteCRM. Also contains some SuiteCRM code to maintain compatibility with SugarCRM. However this is likely to change in the future.

`data` Stores the classes and files used to deal with SugarBeans and their relationships.

`examples`
: Contains a few basic examples of lead capture and API usage. However these are very outdated.

`include`
: Contains the bulk of non module and non data SuiteCRM code.

`install`
: Code used by the SuiteCRM installer.

`jssource`
: The `jssource` folder contains the unminified source of some of the JavaScript files used within SuiteCRM.

`metadata`
: Stores relationship metadata for the various stock SuiteCRM modules. This should not be confused with module metadata which contains information on view, dashlet and search definitions.

`mobile`
: Stores code for the QuickCRM[1] mobile app.

`ModuleInstall`
: Code for the module installer.

`modules`
: Contains the code for any stock or custom SuiteCRM modules.

[1]http://www.quickcrm.fr

`service`
: Code for the SuiteCRM Soap and REST APIs.

`themes`
: Code, data and images for the bundled SuiteCRM theme.

`upload`
: The `upload` folder contains documents that have been uploaded to SuiteCRM. The names of the files comes from the ID of the matching Document Revision/Note. `upload/upgrades` will also contain various upgrade files and the packages of installed modules.

`log4php`, `soap`, `XTemplate`, `Zend`
: Source code for various libraries used by SuiteCRM some of which are deprecated.

3. Working with Beans

Beans are the Model in SuiteCRM's MVC (Model View Controller) architecture. They allow retrieving data from the database as objects and allow persisting and editing records. This section will go over the various ways of working with beans.

BeanFactory

The BeanFactory allows dynamically loading bean instances or creating new records. For example to create a new bean you can use:

Example 3.1: Creating a new Bean using the BeanFactory

```php
$bean = BeanFactory::newBean('<TheModule>');
//For example a new account bean:
$accountBean = BeanFactory::newBean('Accounts');
```

Retrieving an existing bean can be achieved in a similar manner:

Example 3.2: Retrieving a bean with the BeanFactory

```php
$bean = BeanFactory::getBean('<TheModule>', $beanId);
//For example to retrieve an account id
$bean = BeanFactory::getBean('Accounts', $beanId);
```

`getBean` will return an unpopulated bean object if `$beanId` is not supplied or if there's no such record. Retrieving an unpopulated bean can be useful if you wish to use the static methods of the bean (for example see the Searching for Beans section). To deliberately retrieve an unpopulated bean you can omit the second argument of the `getBean` call. I.e.

Example 3.3: Retrieving an unpopulated bean

```php
$bean = BeanFactory::getBean('<TheModule>');
```

`BeanFactory::getBean` caches ten results. This can cause odd behaviour if you call `getBean` again and get a cached copy. Any calls that return a cached copy will return the same instance. This means changes to one of the beans will be reflected in all the results.

Using BeanFactory ensures that the bean is correctly set up and the necessary files are included etc.

SugarBean

The SugarBean is the parent bean class and all beans in SuiteCRM extend this class. It provides various ways of retrieving and interacting with records.

Searching for beans

The following examples show how to search for beans using a bean class. The examples provided assume that an account bean is available names $accountBean. This may have been retrieved using the getBean call mentioned in the BeanFactory section e.g.

Example 3.4: Retrieving an unpopulated account bean

```
$accountBean = BeanFactory::getBean('Accounts');
```

get_list

The get_list method allows getting a list of matching beans and allows paginating the results.

Example 3.5: get_list method signature

```
get_list(
    $order_by = "",
    $where = "",
    $row_offset = 0,
    $limit=-1,
    $max=-1,
    $show_deleted = 0)
```

$order_by
: Controls the ordering of the returned list. `$order_by` is specified as a string that will be used in the SQL ORDER BY clause e.g. to sort by name you can simply pass `name`, to sort by date_-entered descending use `date_entered DESC`. You can also sort by multiple fields. For example sorting by date_modified and id descending `date_modified, id DESC`.

$where
: Allows filtering the results using an SQL WHERE clause. `$where` should be a string containing the SQL conditions. For example in the contacts module searching for contacts with specific first names we might use `contacts.first_name='Jim'`. Note that we specify the table, the query may end up joining onto other tables so we want to ensure that there is no ambiguity in which field we target.

$row_offset
The row to start from. Can be used to paginate the results.

$limit
The maximum number of records to be returned by the query. `-1` means no limit.

$max
The maximum number of entries to be returned per page. `-1` means the default max (usually 20).

$show_deleted
Whether to include deleted results.

Results

get_list will return an array. This will contain the paging information and will also contain the list of beans. This array will contain the following keys:

list An array of the beans returned by the list query

row_count
The total number of rows in the result

next_offset
The offset to be used for the next page or `-1` if there are no further pages.

previous_offset
The offset to be used for the previous page or `-1` if this is the first page.

current_offset
The offset used for the current results.

Example

Let's look at a concrete example. We will return the third page of all accounts with the industry `Media` using `10` as a page size and ordered by name.

Example 3.6: Example get_list call

```
$beanList = $accountBean->get_list(
                                //Order by the accounts name
                                'name',
                                //Only accounts with industry 'Media'
                                "accounts.industry = 'Media'",
                                //Start with the 30th record (third page)
                                30,
                                //No limit - will default to max page size
                                -1,
                                //10 items per page
                                10);
```

This will return:

Example 3.7: Example get_list results

```
Array
(
    //Snipped for brevity - the list of Account SugarBeans
    [list] => Array()
    //The total number of results
    [row_count] => 36
    //This is the last page so the next offset is -1
    [next_offset] => -1
    //Previous page offset
    [previous_offset] => 20
    //The offset used for these results
    [current_offset] => 30
)
```

get_full_list

`get_list` is useful when you need paginated results. However if you are just interested in getting a list of all matching beans you can use `get_full_list`. The `get_full_list` method signature looks like this:

Example 3.8: get_full_list method signature

```
get_full_list(
            $order_by = "",
            $where = "",
            $check_dates=false,
            $show_deleted = 0
```

These arguments are identical to their usage in `get_list` the only difference is the `$check_dates` argument. This is used to indicate whether the date fields should be converted to their display values (i.e. converted to the users date format).

Results

The get_full_list call simply returns an array of the matching beans

Example

Let's rework our `get_list` example to get the full list of matching accounts:

Example 3.9: Example get_full_list call

```
$beanList = $accountBean->get_full_list(
                                //Order by the accounts name
                                'name',
                                //Only accounts with industry 'Media'
                                "accounts.industry = 'Media'"
                                );
```

retrieve_by_string_fields

Sometimes you only want to retrieve one row but may not have the id of the record. `retrieve_by_string_fields` allows retrieving a single record based on matching string fields.

Example 3.10: retrieve_by_string_fields method signature

```
retrieve_by_string_fields(
                          $fields_array,
                          $encode=true,
                          $deleted=true)
```

$fields_array
An array of field names to the desired value.

$encode
Whether or not the results should be HTML encoded.

$deleted
Whether or not to add the deleted filter.

Note here that, confusingly, the deleted flag works differently to the other methods we have looked at. It flags whether or not we should filter out deleted results. So if true is passed then the deleted results will *not* be included.

Results

retrieve_by_string_fields returns a single bean as it's result or null if there was no matching bean.

Example

For example to retrieve the account with name `Tortoise Corp` and account_type `Customer` we could use the following:

Example 3.11: Example retrieve_by_string_fields call

```
$beanList = $accountBean->retrieve_by_string_fields(
                                array(
                                  'name' => 'Tortoise Corp',
                                  'account_type' => 'Customer'
                                )
                              );
```

Accessing fields

If you have used one of the above methods we now have a bean record. This bean represents the record that we have retrieved. We can access the fields of that record by simply accessing properties on the bean just like any other PHP object. Similarly we can use property access to set the values of beans. Some examples are as follows:

Example 3.12: Accessing fields examples

```
//Get the Name field on account bean
$accountBean->name;

//Get the Meeting start date
$meetingBean->date_start;

//Get a custom field on a case
$caseBean->third_party_code_c;

//Set the name of a case
$caseBean->name = 'New Case name';

//Set the billing address post code of an account
$accountBean->billing_address_postalcode = '12345';
```

When changes are made to a bean instance they are not immediately persisted. We can save the changes to the database with a call to the beans `save` method. Likewise a call to `save` on a brand new bean will add that record to the database:

Example 3.13: Persisting bean changes

```
//Get the Name field on account bean
$accountBean->name = 'New account name';
//Set the billing address post code of an account
$accountBean->billing_address_postalcode = '12345';
//Save both changes.
$accountBean->save();

//Create a new case (see the BeanFactory section)
$caseBean = BeanFactory::newBean('Cases');
//Give it a name and save
$caseBean->name = 'New Case name';
$caseBean->save();
```

Whether to save or update a bean is decided by checking the `id` field of the bean. If `id` is set then SuiteCRM will attempt to perform an update. If there is no `id` then one will be generated and a new record will be inserted into the database. If for some reason you have supplied an `id` but the record is new (perhaps in a custom import script) then you can set `new_with_id` to true on the bean to let SuiteCRM know that this record is new.

Related beans

We have seen how to save single records but, in a CRM system, relationships between records are as important as the records themselves. For example an account may have a list of cases associated with it, a contact will have an account that it falls under etc. We can get and set relationships between beans using several methods.

get_linked_beans

The `get_linked_beans` method allows retrieving a list of related beans for a given record.

Example 3.14: get_linked_beans method signature

```
get_linked_beans(
                $field_name,
                $bean_name,
                $sort_array = array(),
                $begin_index = 0,
                $end_index = -1,
                $deleted=0,
                $optional_where="");
```

$field_name
: The link field name for this link. Note that this is not the same as the name of the relationship. If you are unsure of what this should be you can take a look into the cached vardefs of a module in `cache/modules/<TheModule>/<TheModule>Vardefs.php` for the link definition.

$bean_name
: The name of the bean that we wish to retrieve.

$sort_array
: This is a legacy parameter and is unused.

$begin_index
: Skips the initial `$begin_index` results. Can be used to paginate.

$end_index
: Return up to the `$end_index` result. Can be used to paginate.

$deleted
: Controls whether deleted or non deleted records are shown. If true only deleted records will be returned. If false only non deleted records will be returned.

$optional_where
: Allows filtering the results using an SQL WHERE clause. See the `get_list` method for more details.

Results

`get_linked_beans` returns an array of the linked beans.

Example

Example 3.15: Example get_linked_beans call

```
$accountBean->get_linked_beans(
                'contacts',
                'Contacts',
                array(),
                0,
                10,
                0,
                "contacts.primary_address_country = 'USA'");
```

relationships

In addition to the `get_linked_beans` call you can also load and access the relationships more directly.

Loading

Before accessing a relationship you must use the `load_relationship` call to ensure it is available. This call takes the link name of the relationship (not the name of the relationship). As mentioned previously you can find the name of the link in `cache/modules/<TheModule>/<TheModule>Vardefs.php` if you're not sure.

Example 3.16: Loading a relationship

```
//Load the relationship
$accountBean->load_relationship('contacts');
//Can now call methods on the relationship object:
$contactIds = $accountBean->contacts->get();
```

Methods

`get` Returns the ids of the related records in this relationship e.g for the account - contacts relationship in the example above it will return the list of ids for contacts associated with the account.

`getBeans` Similar to `get` but returns an array of beans instead of just ids.

⚠ `getBeans` will load the full bean for each related record. This may cause poor performance for relationships with a large number of beans.

`add` Allows relating records to the current bean. `add` takes a single id or bean or an array of ids or beans. If the bean is available this should be used since it prevents reloading the bean. For example to add a contact to the relationship in our example we can do the following:

Example 3.18: Adding a new contact to a relationship

```
//Load the relationship
$accountBean->load_relationship('contacts');

//Create a new demo contact
$contactBean = BeanFactory::newBean();
$contactBean->first_name = 'Jim';
$contactBean->last_name = 'Mackin';
$contactBean->save();

//Link the bean to $accountBean
$accountBean->contacts->add($contactBean);
```

`delete` `delete` allows unrelating beans. Counter-intuitively it accepts the ids of both the bean and the related bean. For the related bean you should pass the bean if it is available e.g when unrelating an account and contact:

Example 3.19: Removing a new contact from a relationship

```
//Load the relationship
$accountBean->load_relationship('contacts');

//Unlink the contact from the account - assumes $contactBean is a Contact SugarB\
ean
$accountBean->contacts->delete($accountBean->id, $contactBean);
```

⚠ Be careful with the delete method. Omitting the second argument will cause all relationships for this link to be removed.

4. Vardefs

What are Vardefs

The Vardefs are used to supply information to SuiteCRM about a particular bean. These generally specify the fields, relationships and indexes in a given module as well as additional information such as whether it is audited, the table name etc.

Defining Vardefs

Module

Vardefs are initially defined in their respective modules folder. For the Accounts module this will be in modules/Accounts/vardefs.php. The information is stored in an array named $dictionary using the module name as the key. For Accounts this will be `$dictionary['Account']`. Let’s look at the Account vardefs (which have been edited for brevity):

Example 4.1: Account Vardefs

```php
$dictionary['Account'] =
array(
        'table' => 'accounts',
        'audited'=>true,
        'unified_search' => true,
        'unified_search_default_enabled' => true,
        'duplicate_merge'=>true,
        'comment' => 'Accounts are organizations or entities that ...',
        'fields' => array (
          //Snipped for brevity. See the fields section.
        ),
        'indices' => array (
          //Snipped for brevity. See the indices section.
        ),
        'relationships' => array (
          //Snipped for brevity. See the relationship section.
        ),
        //This enables optimistic locking for Saves From EditView
        'optimistic_locking'=>true,
);

VardefManager::createVardef(
        'Accounts',
        'Account',
        array('default', 'assignable','company',)
);
```

Keys

The following are some of the keys that can be specified for the vardefs. Fields, indices and relationships are covered in their own sections.

`table`
: The database table name for this module.

`audited`
: Whether or not this module should be audited. Note that `audited` must also be set at the fields level for a field to be audited.

`unified_search`
: Whether this module can be searchable via the global search.

`unified_search_default_enabled`
: Whether this module is searchable via the global search by default.

`duplicate_merge`
: Whether or not duplicate merging functionality is enabled for this module.

`comment`
: A description of this module.

`optimistic_locking`
: Whether optimistic should be enabled for this module. Optimistic locking locks concurrent edits on a record by assuming that there will be no conflict. On save the last modified timestamp on the record will be checked. If it is different then an edit has occurred since this record was loaded. If this is the case then the user will be prompted with a page showing the differences in the two edits and asked to choose which edits are to be used.

Fields

The field defines the behaviour and attributes of each field in the module.

`name` The name of the field.

`vname`
: The name of the language label to be used for this field.

`type`
: The type of the field. See the field types section.

`isnull`
: Whether null values are allowed

`len`
: If the field is a string type, the max number of characters allowed.

`options`
: For enum fields the language label for the dropdown values for this field

`dbtype`
: The type to be used by the database to store this field. This is not required as the appropriate type is usually chosen.

`default`
: The default value of this field.

`massupdate`
: Whether or not this field should be mass updatable. Note that some field types are always restricted from mass updates.

`rname`
: For related fields only. The name of the field to be taken from the related module.

`id_name`
: For related fields only. The field in this bean which contains the related id.

`source`
: The source of this field. Can be set to ‘non-db’ if the field is not stored in the database - for example for link fields, fields populated by logic hooks or by other means.

`sort_on`
: For concatenated fields (i.e. name fields) the field which should be used to sort.

`fields`
: For concatenated fields (i.e. name fields) an array of the fields which should be concatenated.

`db_concat_fields`
: For concatenated fields (i.e. name fields) an array of the fields which should be concatenated in the database. Usually this is the same as fields.

`unified_search`
: True if this field should be searchable via the global search.

`enable_range_search`
: Whether the list view search should allow a range search of this field. This is used for date and numeric fields.

`studio`
: Whether the field should display in studio.

`audited`
: Whether or not changes to this field should be audited.

Field types

The following are common field types used:

`id` An id field.

`name` A name field. This is usually a concatenation of other fields.

`bool` A boolean field.

`varchar`
: A variable length string field.

`char` A character field.

`text` A text area field.

`decimal`
: A decimal field.

`date` A date field.

`datetime`
: A date and time field.

`enum` A dropdown field.

`phone`
: A phone number field.

`link` A link to another module via a relationship.

`relate`
: A related bean field.

Indices

The indices array allows defining any database indexes that should be in place on the database table for this module. Let's look at an example:

Example 4.2: Example indices definition

```
'indices' => array (
        array(
                'name' =>'idx_mymod_id_del',
                'type' =>'index',
                'fields'=>array('id', 'deleted')),
        array(
                'name' =>'idx_mymod_parent_id',
                'type' =>'index',
                'fields'=>array( 'parent_id')),
        array(
                'name' =>'idx_mymod_parent_id',
                'type' =>'unique',
                'fields'=>array( 'third_party_id')),
        ),
```

Each array entry should have, at least, the following entries:

name
: The name of the index. This is usually used by the database to reference the index. Most databases require that these are unique.

type
: The type of the index to create. `index` will simply add an index on the fields, `unique` will add a unique constraint on the fields, `primary` will add the fields as a primary key.

fields
: An array of the fields to be indexed. The order of this array will be used as the order of the fields in the index.

Relationships

The Vardefs also specify the relationships within this module. Here's an edited example from the Accounts module:

Example 4.3: Example relationships definition

```
'relationships' => array (
        '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'),
),
```

Here we see the link between accounts and cases. This is specified with the following keys:

`lhs_module`
: The module on the left hand side of this relationship. For a one to many relationship this will be the “One” side.

`lhs_table`
: The table for the left hand side module. If you are unsure the table for a module can be found in it’s vardefs.

`lhs_key`
: The field to use for the left hand side of this link. In this case it is the `id` of the account.

`rhs_module`
: The right hand side module. In this case the “many” side of the relationship.

`rhs_table`
: The table for the right hand side module. As stated previously you can find the table for a module can be found in it’s vardefs.

`rhs_key`
: The field to use on the right hand side. In this case the `account_id` field on cases.

`relationship_type`
: The type of relationship - “one-to-many” or “many-to-many”. Since this is a one to many relationship it means a case is related to a single account but a single account can have multiple cases.

For many to many relationship fields the following keys are also available:

`join_table`
: The name of the join table for this relationship.

`join_key_lhs`
: The name of the field on the join table for the left hand side.

`join_key_rhs`
: The name of the field on the join table for the right hand side.

Vardef templates

Vardef templates provide a shortcut for defining common vardefs. This is done by calling `VardefManager::createVardef` and passing the module name, object name and an array of templates to be assigned. The following is an example from the accounts vardefs:

Example 4.4: Example vardef template

```
VardefManager::createVardef(
                'Accounts',
                'Account',
                array('default', 'assignable','company',)
                );
```

In this example the `default`, `assignable` and `company` templates are used. The following are some of the available templates:

`basic`

`default`
: Adds the common base fields such as `id`, `name`, `date_entered`, etc.

`assignable`
: Adds the fields and relationships necessary to assign a record to a user.

`person`
: Adds fields common to people records such as `first_name`, `last_name`, address, etc.

`company`
: Adds fields common to companies such as an industry dropdown, address, etc.

Customising vardefs

Vardefs can be customised by adding a file into

Example 4.5: Custom vardef location

```
custom/Extension/modules/<TheModule>/Ext/SomeFile.php
```

This file can then be used to add a new field definition or customise an existing one e.g changing a field type:

Example 4.6: Example overriding an existing vardef

```
$dictionary["TheModule"]["fields"]["some_field"]['type'] = 'int';
```

5. Views

SuiteCRM follows the MVC (Model-View-Controller) pattern and as such has the concept of views. Views are responsible for gathering and displaying data . There are a number of default views in SuiteCRM. These include

ListView
: Displays a list of records and provides links to the EditViews and DetailViews of those records. The ListView also allows some operations such as deleting and mass updating records. This is (usually) the default view for a module.

DetailView
: Displays the details of a single record and also displays subpanels of related records.

EditView
: The EditView allows editing the various fields of a record and provides validation on these values.

Location

Views can be found in `modules/<TheModule>/views/` or, for custom views, `custom/modules/<TheModule>/views/`, and are named in the following format: `view.<viewname>.php`. For example, the Accounts DetailView can be found in `modules/Accounts/views/view.detail.php` with a customised version in `custom/modules/Accounts/views/view.detail.php`. The custom version is used if it exists. If it doesn't then the module version is used. Finally, if neither of these exist then the SuiteCRM default is used in `include/MVC/View/views/`.

Customising

In order to customise a View we first need to create the appropriate view file. This will vary depending on the module we wish to target.

Custom module

In this case we can place the file directly into our module. Create a new file (if it doesn't exist) at `modules/<TheModule>/views/view.<viewname>.php`. The contents will look similar to:

Example 5.1: View for a custom module

```
<?php

require_once 'include/MVC/View/views/view.<viewname>.php';

if(!defined('sugarEntry') || !sugarEntry) die('Not A Valid Entry Point');
class <TheModule>View<ViewName> extends View<ViewName>
{

}
```

A more concrete example would be for the detail view for a custom module called ABC_Vehicles:

Example 5.2: Detail view for a custom module, ABC_Vehicles

```
<?php

require_once 'include/MVC/View/views/view.detail.php';

if(!defined('sugarEntry') || !sugarEntry) die('Not A Valid Entry Point');
class ABC_VehiclesViewDetail extends ViewDetail
{

}
```

Preexisting modules

For preexisting modules you will want to add the view to `custom/modules/<TheModule>/views/view.<viewname>.php`.

The contents of this file will vary depending on whether you wish to extend the existing view (if it exists) or create your own version completely. It is usually best to extend the existing view, since this will retain important logic. Note the naming convention here. We name the class `Custom<TheModule>View<ViewName>` (for example `CustomAccountsViewDetail`).

Here we don't extend the existing view or no such view exists:

Example 5.3: Custom view for an existing module

```php
<?php
if(!defined('sugarEntry') || !sugarEntry) die('Not A Valid Entry Point');

require_once 'include/MVC/View/views/view.<viewname>.php';

class Custom<TheModule>View<ViewName> extends ViewDetail
{

}
```

Otherwise we extend the existing view. Note that we are requiring the existing view:

Example 5.4: Overriding a view for an existing module

```php
<?php
if(!defined('sugarEntry') || !sugarEntry) die('Not A Valid Entry Point');

require_once 'modules/<TheModule>/views/view.<viewname>.php';

class Custom<TheModule>View<ViewName> extends <TheModule>View<ViewName>
{

}
```

For example, overriding the List View of Accounts:

Example 5.5: Overriding the Accounts List View

```php
<?php
if(!defined('sugarEntry') || !sugarEntry) die('Not A Valid Entry Point');

require_once 'modules/Accounts/views/view.list.php';

class CustomAccountsViewList extends AccountsViewList
{

}
```

Making changes

Now that we have a custom view what can we actually do? The views have various methods which we can now override to change/add behaviour. The most common ones to override are:

preDisplay
: Explicitly intended to allow logic to be called before display() is called. This can be used to alter arguments to the list view or to output anything to appear before the main display code (such as, for example, adding JavaScript).

display
: Does the actual work of displaying the view. Can be overridden to alter this behaviour or to output anything after the main display. You usually want to call parent::display(); to ensure that the display code is run (unless, of course, you are adding your own display logic).

6. Metadata

Intro

Module metadata are used to describe how various views behave in the module. The main use of this is providing field and layout information but this can also be used to filter subpanels and to describe what fields are used in the search.

Location

Module metadata can be found in:

Example 6.1: Module metadata location

```
modules/<TheModule>/metadata/
```

Customising

Usually studio is the best way of customising metadata. Even when you do wish to make customisations that are not possible through studio it can be simpler to set everything up in studio first. This is particularly true for layout based metadata. However if you are customising metadata it is as simple as placing, or editing, the file in the custom directory. For example to override the Accounts detailviewdefs (found in `modules/Accounts/metadata/detailviewdefs.php`) we would place (or edit) the file in `custom/modules/Accounts/metadata/detailviewdefs.php`. One exception to this rule is the studio.php file. The modules metadata folder is the only location checked - any version in `custom/<TheModule>/metadata/studio.php` is ignored.

Different metadata

detailviewdefs.php

detailviewdefs.php provides information on the layout and fields of the detail view for this module. This file uses the same structure as editviewdefs.php. Let’s look at an example for a fictional module `ABC_Vehicles`:

Example 6.2: DetailView metadata definition

```
<?php
$viewdefs ['ABC_Vehicles'] ['DetailView'] = array (
        'templateMeta' => array (
                'form' => array (
                        'buttons' => array (
                                'EDIT',
                                'DUPLICATE',
                                'DELETE',
                                'FIND_DUPLICATES'
                        )
                ),
                'maxColumns' => '2',
                'widths' => array (
                        array (
                                'label' => '10',
                                'field' => '30'
                        ),
                        array (
                                'label' => '10',
                                'field' => '30'
                        )
                ),
                'includes' => array (
                        array (
                                'file' => 'modules/ABC_Vehicles/ABC_VehiclesDetail.js'
                        )
                )
        ),
        'panels' => array (
                'LBL_ABC_VEHICLES_INFO' => array (
                        array (
                                array (
                                        'name' => 'name',
                                        'comment' => 'The Name of the Vehicle',
                                        'label' => 'LBL_NAME',
                                ),
                                'reg_number'
                        ),
                        array (
                                array (
                                        'name' => 'type',
42                                          'label' => 'LBL_TYPE',
43                                  ),
44                                  array (
45                                          'name' => 'phone_fax',
46                                          'comment' => 'The fax phone number of this company',
47                                          'label' => 'LBL_FAX'
48                                  )
49                          ),
50                          array (
51                                  array (
52                                          'name' => 'registered_address_street',
53                                          'label' => 'LBL_REGISTERED_ADDRESS',
54                                          'type' => 'address',
55                                          'displayParams' => array (
56                                                  'key' => 'registered'
57                                          )
58                                  ),
59                          ),
60                  ),
61                  'LBL_PANEL_ADVANCED' => array (
62        array (
63                                  array (
64                                          'name' => 'assigned_user_name',
65                                          'label' => 'LBL_ASSIGNED_TO'
66                                  ),
67                                  array (
68                                          'name' => 'date_modified',
69                                          'label' => 'LBL_DATE_MODIFIED',
70                                          'customCode' => '{$fields.date_modified.value} '
71                                                          + '{$APP.LBL_BY} '
72                                                          + '{$fields.modified_by_name.value}',
73                                  )
74                          ),
75                  ),
76          )
77  );
78  ?>
```

We see that line 2 defines an array `$viewdefs['ABC_Vehicles']['DetailView']` which places a DetailView entry for the module ABC_Vehicles into `$viewdefs` (DetailView will be EditView or QuickCreateView as appropriate). This array has two main keys defined here:

templateMeta

The templateMeta key provides information about the view in general. The `['form']['buttons']` entries define the buttons that should appear in this view.

`maxColumns`
Defines the number of columns to use for this view. It is unusual for this to be more than 2.

`widths`
An array defining the width of the label and field for each column.

`includes`
An array of additional JavaScript files to include. This is useful for adding custom JavaScript behaviour to the page.

panels

The panels entry defines the actual layout of the Detail (or Edit) view. Each entry is a new panel in the view with the key being the label for that panel. We can see in our example that we have 2 panels. One uses the label defined by the language string `LBL_ABC_VEHICLES_INFO`, the other uses `LBL_PANEL_ADVANCED`.

Each panel has an array entry for each row, with each array containing an entry for each column. For example we can see that the first row has the following definition:

Example 6.3: DetailView metadata row definition

```
array(
        array (
                'name' => 'name',
                'comment' => 'The Name of the Vehicle',
                'label' => 'LBL_NAME',
        ),
        'reg_number',
),
```

This has an array definition for the first row, first column and a string definition for the first row, second column. The string definition is very straightforward and simply displays the detail (or edit, as appropriate) view for that field. It will use the default label, type, etc. In our example we are displaying the field named `reg_number`.

The array definition for the first row, first column is a little more complex. Each array definition must have a `name` value. In our example we are displaying the `name` field. However we also supply some other values. Values most commonly used are:

`comment`
: Used to note the purpose of the field.

`label`
: The language key for this label. If the language key is not recognised then this value will be used instead (see the chapter on language).

`displayParams`
: An array used to pass extra arguments for the field display. For the options and how they are used you can have a look into the appropriate field type in `include/SugarFields/Fields` or `custom/include/SugarFields/Fields`. An example is setting the size of a textarea:

Example 6.4: DetailView metadata displayParams

```php
'displayParams' => array(
    'rows' => 2,
    'cols' => 30,
),
```

customCode
: Allows supplying custom smarty code to be used for the display. The code here can include any valid smarty code and this will also have access to the current fields in this view via `$fields`. An example of outputing the ID field would be `{$fields.id.value}`. Additionally the module labels and app labels can be accessed via `$MOD` and `$APP` respectively. Finally you can use `@@FIELD@@` to output the value of the field that would have been used. For example `{if $someCondition}@@FIELD@@{/if}` will conditionally show the field.

editviewdefs.php

`editviewdefs.php` provides information on the layout and fields of the edit view for this module. This file uses the same structure as detailviewdefs.php. Please see the information on detailviewdefs.php.

listviewdefs.php

The `listviewdefs.php` file for a module defines what fields the list view for that module will display. Let’s take a look at an example:

Example 6.5: ListView metadata definition

```
$listViewDefs ['AOR_Reports'] =
array (
  'NAME' =>
  array (
    'width' => '15%',
    'label' => 'LBL_NAME',
    'default' => true,
    'link' => true,
  ),
  'REPORT_MODULE' =>
  array (
    'type' => 'enum',
    'default' => true,
    'studio' => 'visible',
    'label' => 'LBL_REPORT_MODULE',
    'width' => '15%',
  ),
  'ASSIGNED_USER_NAME' =>
  array (
    'width' => '15%',
    'label' => 'LBL_ASSIGNED_TO_NAME',
    'module' => 'Employees',
    'id' => 'ASSIGNED_USER_ID',
    'default' => true,
  ),
  'DATE_ENTERED' =>
  array (
    'type' => 'datetime',
    'label' => 'LBL_DATE_ENTERED',
    'width' => '15%',
    'default' => true,
  ),
  'DATE_MODIFIED' =>
  array (
    'type' => 'datetime',
    'label' => 'LBL_DATE_MODIFIED',
    'width' => '15%',
    'default' => true,
  ),
);
```

To define the list view defs we simply add a key to the `$listViewDefs` array. In this case we add an entry for `AOR_Reports` This array contains an entry for each field that we wish to show in the list view and is keyed by the upper case name of the field. For example, the `REPORT_MODULE` key refers to the `report_module` field of AOR_Reports.

type The type of the field. This can be used to override how a field is displayed.

default
Whether this field should be shown in the list view by default. If false then the field will appear in the available columns list in studio.

studio
Whether or not this field should be displayed in studio. This can be useful to ensure that a critical field is not removed.

label
The label to be used for this field. If this is not supplied then the default label for that field will be used.

width
The width of the field in the list view. Note that, although this is usually given as a percentage it is treated as a proportion. The example above has five columns with a width of `15%` but these will actually be `20%` since this is a ratio.

popupdefs.php

popupdefs.php provides information on the layout, fields and search options of the module popup that is usually used when selecting a related record.

Let's look at the default popupdefs.php for the Accounts module:

Example 6.6: PopupView metadata definition

```
$popupMeta = array(
        'moduleMain' => 'Case',
        'varName' => 'CASE',
        'className' => 'aCase',
        'orderBy' => 'name',
        'whereClauses' =>
                array('name' => 'cases.name',
                                'case_number' => 'cases.case_number',
                                'account_name' => 'accounts.name'),
        'listviewdefs' => array(
                'CASE_NUMBER' => array(
                        'width' => '5',
                        'label' => 'LBL_LIST_NUMBER',
                'default' => true),
                'NAME' => array(
                        'width' => '35',
                        'label' => 'LBL_LIST_SUBJECT',
                        'link' => true,
                'default' => true),
                'ACCOUNT_NAME' => array(
                        'width' => '25',
                        'label' => 'LBL_LIST_ACCOUNT_NAME',
                        'module' => 'Accounts',
                        'id' => 'ACCOUNT_ID',
                        'link' => true,
                'default' => true,
                'ACLTag' => 'ACCOUNT',
                'related_fields' => array('account_id')),
                'PRIORITY' => array(
                        'width' => '8',
                        'label' => 'LBL_LIST_PRIORITY',
                'default' => true),
                'STATUS' => array(
                        'width' => '8',
                        'label' => 'LBL_LIST_STATUS',
                'default' => true),
            'ASSIGNED_USER_NAME' => array(
                'width' => '2',
                'label' => 'LBL_LIST_ASSIGNED_USER',
                'default' => true,
               ),
                ),
        'searchdefs'   => array(
                 'case_number',
                'name',
                array(
                        'name' => 'account_name',
                        'displayParams' => array(
                                'hideButtons'=>'true',
                                'size'=>30,
                                'class'=>'sqsEnabled sqsNoAutofill'
                        )
                ),
                'priority',
                'status',
                array(
                        'name' => 'assigned_user_id',
                        'type' => 'enum',
                        'label' => 'LBL_ASSIGNED_TO',
                        'function' => array(
                                'name' => 'get_user_array',
                                'params' => array(false))
                        ),
          )
);
```

The popupdefs.php specifies a `$popupMeta` array with the following keys:

`moduleMain`
: The module that will be displayed by this popup.

`varName`
: The variable name used to store the search preferences etc. This will usually simply the upper case module name.

`className`
: The class name of the SugarBean for this module. If this is not supplied then `moduleMain` will be used. This is only really required for classes where the class name and module name differ (such as Cases).

`orderBy`
: The default field the list of records will be sorted by.

`whereClauses`
: Legacy option. This is only used as a fallback when there are no searchdefs. Defines the names of fields to allow searching for and their database representation.

`listviewdefs`
: The list of fields displayed in the popup list view. See `listviewdefs.php`.

`searchdefs`
: An array of the fields that should be available for searching in the popup. See the individual search defs in the searchdefs.php section (for example the `basic_search` array).

quickcreatedefs.php

`quickcreatedefs.php` provides information on the layout and fields of the quick create view for this module (this is the view that appears when creating a record from a subpanel). This file uses the same structure as `detailviewdefs.php`. Please see the information on `detailviewdefs.php`.

searchdefs.php

The search defs of a module define how searching in that module looks and behaves.

Let's look at an example.

Example 6.7: Search View metadata definition

```php
$searchdefs ['Accounts'] = array (
        'templateMeta' => array (
                'maxColumns' => '3',
                'maxColumnsBasic' => '4',
                'widths' => array (
                        'label' => '10',
                        'field' => '30'
                )
        ),
        'layout' => array (
                'basic_search' => array (
                        'name' => array (
                                'name' => 'name',
                                'default' => true,
                                'width' => '10%'
                        ),
                        'current_user_only' => array (
                                'name' => 'current_user_only',
                                'label' => 'LBL_CURRENT_USER_FILTER',
                                'type' => 'bool',
                                'default' => true,
                                'width' => '10%'
                        )
                )
                ,
                'advanced_search' => array (
                        'name' => array (
                                'name' => 'name',
                                'default' => true,
                                'width' => '10%'
                        ),
                        'website' => array (
                                'name' => 'website',
                                'default' => true,
                                'width' => '10%'
                        ),
                        'phone' => array (
                                'name' => 'phone',
                                'label' => 'LBL_ANY_PHONE',
                                'type' => 'name',
                                'default' => true,
                                'width' => '10%'
                        ),
                        'email' => array (
                                'name' => 'email',
                                'label' => 'LBL_ANY_EMAIL',
                                'type' => 'name',
                                'default' => true,
                                'width' => '10%'
                        ),
                        'address_street' => array (
                                'name' => 'address_street',
                                'label' => 'LBL_ANY_ADDRESS',
                                'type' => 'name',
                                'default' => true,
                                'width' => '10%'
                        ),
                        'address_city' => array (
                                'name' => 'address_city',
                                'label' => 'LBL_CITY',
                                'type' => 'name',
                                'default' => true,
                                'width' => '10%'
                        ),
                        'address_state' => array (
                                'name' => 'address_state',
                                'label' => 'LBL_STATE',
                                'type' => 'name',
                                'default' => true,
                                'width' => '10%'
                        ),
                        'address_postalcode' => array (
                                'name' => 'address_postalcode',
                                'label' => 'LBL_POSTAL_CODE',
                                'type' => 'name',
                                'default' => true,
                                'width' => '10%'
                        ),
                        'billing_address_country' => array (
                                'name' => 'billing_address_country',
                                'label' => 'LBL_COUNTRY',
                                'type' => 'name',
                                'options' => 'countries_dom',
                                'default' => true,
                                'width' => '10%'
                        ),
                        'account_type' => array (
                                'name' => 'account_type',
                                'default' => true,
                                'width' => '10%'
                        ),
                        'industry' => array (
                                'name' => 'industry',
                                'default' => true,
                                'width' => '10%'
                        ),
                        'assigned_user_id' => array (
                                'name' => 'assigned_user_id',
                                'type' => 'enum',
                                'label' => 'LBL_ASSIGNED_TO',
                                'function' => array (
                                        'name' => 'get_user_array',
                                        'params' => array (
                                                        0 => false
                                        )
                                ),
                                'default' => true,
                                'width' => '10%'
                        )
                )
        )
);
```

Here we setup a new array for `Accounts` in the `$searchdefs` array. This has two keys:

templateMeta

The `templateMeta` key controls the basic look of the search forms. Here we define some overall layout info such as the maximum columns (3) and the maximum number of columns for the basic search (4). Finally we set the widths for the search fields and their labels.

layout

The `layout` key contains the layout definitions for the basic search and advanced search. This is simply a list of array definition of the fields. See the section on listviewdefs.php for a description of some of the options.

`subpaneldefs.php`

The subpaneldefs.php file provides definitions for the subpanels that appear in the detail view of a module. Let's look at an example:

Example 6.8: Subpanel metadata definition

```php
$layout_defs['AOS_Quotes'] = array (
        'subpanel_setup' => array (
                'aos_quotes_aos_contracts' => array (
                        'order' => 100,
                        'module' => 'AOS_Contracts',
                        'subpanel_name' => 'default',
                        'sort_order' => 'asc',
                        'sort_by' => 'id',
                        'title_key' => 'AOS_Contracts',
                        'get_subpanel_data' => 'aos_quotes_aos_contracts',
                        'top_buttons' => array (
                                0 => array (
                                        'widget_class' => 'SubPanelTopCreateButton'
                                ),
                                1 => array (
                                        'widget_class' => 'SubPanelTopSelectButton',
                                        'popup_module' => 'AOS_Contracts',
                                        'mode' => 'MultiSelect'
                                )
                        )
                ),
                'aos_quotes_aos_invoices' => array (
                        'order' => 100,
                        'module' => 'AOS_Invoices',
                        'subpanel_name' => 'default',
                        'sort_order' => 'asc',
                        'sort_by' => 'id',
                        'title_key' => 'AOS_Invoices',
                        'get_subpanel_data' => 'aos_quotes_aos_invoices',
                        'top_buttons' => array (
                                0 => array (
                                        'widget_class' => 'SubPanelTopCreateButton'
                                ),
                                1 => array (
                                        'widget_class' => 'SubPanelTopSelectButton',
                                        'popup_module' => 'AOS_Invoices',
                                        'mode' => 'MultiSelect'
                                )
                        )
                ),
                'aos_quotes_project' => array (
                        'order' => 100,
                        'module' => 'Project',
                        'subpanel_name' => 'default',
                        'sort_order' => 'asc',
                        'sort_by' => 'id',
                        'title_key' => 'Project',
                        'get_subpanel_data' => 'aos_quotes_project',
                        'top_buttons' => array (
                                0 => array (
                                        'widget_class' => 'SubPanelTopCreateButton'
                                ),
                                1 => array (
                                        'widget_class' => 'SubPanelTopSelectButton',
                                        'popup_module' => 'Accounts',
                                        'mode' => 'MultiSelect'
                                )
                        )
                )
        )
);
```

In the example above we set up a definition for a module (in this case `AOS_Quotes`) in the `$layout_defs` array. This has a single key `subpanel_setup` which is an array of each of the subpanel definitions keyed by a name. This name should be something recognisable. In the case above it is the name of the link field displayed by the subpanel. The entry for each subpanel usually has the following defined:

order
: A number used for sorting the subpanels. The values themselves are arbitrary and are only used relative to other subpanels.

module
: The module which will be displayed by this subpanel. For example the `aos_quotes_project` def in the example above will display a list of `Project` records.

subpanel_name
: The subpanel from the displayed module which will be used. See the subpanels section of this chapter.

sort_by
: The field to sort the records on.

sort_order
: The order in which to sort the `sort_by` field. `asc` for ascending `desc` for descending.

title_key
: The language key to be used for the label of this subpanel.

get_subpanel_data
: Used to specify where to retrieve the subpanel records. Usually this is just a link name for the current module. In this case the related records will be displayed in the subpanel. However, for more complex links, it is possible to specify a function to call. When specifying a function you should ensure that the `get_subpanel_data` entry is in the form `function:theFunctionName`. Additionally you can specify the location of the function and any additional parameters that are needed by using the `function_parameters` key. An example of a subpanel which uses a function can be found in Appendix A.

function_parameters
: Specifies the parameters for a subpanel which gets it's information from a function (see `get_subpanel_data`). This is an array which allows specifying where the function is by using the `import_function_file` key (if this is absent but `get_subpanel_data` defines a function then the function will be called on the bean for the parent of the subpanel). Additionally this array will be passed as an argument to the function defined in `get_subpanel_data` which allows passing in arguments to the function.

generate_select
: For function subpanels (see `get_subpanel_data`) whether or not the function will return an array representing the query to be used (for `generate_select = true`) or whether it will simply return the query to be used as a string.

get_distinct_data

Whether or not to only return distinct rows. Relationships do not allow linking two records more than once therefore this only really applies if the subpanel source is a function. See `get_subpanel_data` for information on function subpanel sources.

top_buttons

Allows defining the buttons to appear on the subpanel. This is simply an array of the button definitions. These definitions have, at least, the `widget_class` defined which decides the button class to use in `include/generic/SugarWidgets`. Depending on the button this array may also be used to pass in extra arguments to the widget class.

subpanels

Inside the metadata folder is the `subpanels` folder. This allows creating different subpanel layouts for different parent modules. For example, the Contacts module will display differently in the subpanel on an account than it will in the subpanel of a case. The files inside the `subpanels` folder can be named anything. All that matters is that it can be referenced in the `subpanel_name` of the `subpaneldefs.php` of the parent module. The usual subpanel file is simply called `default.php`. Let's look at the `modules/Accounts/metadata/subpanels/default.php` file:

Example 6.8: Module Subpanels definition

```
$subpanel_layout = array(
        'top_buttons' => array(
                array(
                        'widget_class' => 'SubPanelTopCreateButton'
                ),
                array(
                        'widget_class' => 'SubPanelTopSelectButton',
                        'popup_module' => 'Accounts'
                ),
        ),
        'where' => '',
        'list_fields' => array (
          'name' =>
          array (
           'vname' => 'LBL_LIST_ACCOUNT_NAME',
           'widget_class' => 'SubPanelDetailViewLink',
           'width' => '45%',
           'default' => true,
          ),
          'billing_address_city' =>
          array (
            'vname' => 'LBL_LIST_CITY',
            'width' => '20%',
            'default' => true,
          ),
          'billing_address_country' =>
          array (
                  'type' => 'varchar',
            'vname' => 'LBL_BILLING_ADDRESS_COUNTRY',
            'width' => '7%',
            'default' => true,
          ),
          'phone_office' =>
          array (
                  'vname' => 'LBL_LIST_PHONE',
                  'width' => '20%',
                  'default' => true,
          ),
          'edit_button' =>
          array (
                  'vname' => 'LBL_EDIT_BUTTON',
                  'widget_class' => 'SubPanelEditButton',
                  'width' => '4%',
                  'default' => true,
          ),
          'remove_button' =>
          array (
            'vname' => 'LBL_REMOVE',
            'widget_class' => 'SubPanelRemoveButtonAccount',
            'width' => '4%',
            'default' => true,
          ),
   )
);
```

There are three keys in the `$subpanel_layout` variable for this subpanel. These are:

`top_buttons`
Defines the buttons that will appear at the top of the subpanel. See the `top_buttons` key in `subpaneldefs.php`.

`where`
Allows the addition of conditions to the `where` clause. For example this could be used to

exclude Cases that are closed (`cases.state != "Closed"`) or only include Accounts of a specific industry (`accounts.industry = "Media"`). Note that in these examples we specify the table to remove any ambiguity in the query.

`list_fields`
Defines the list of fields to be displayed in this subpanel. See the section on `listviewdefs.php` for more information.

studio.php

studio.php is the simplest file in metadata and it's existence is simply used to confirm if a module should be shown in studio for user tweaking. Note that, unlike other metadata files, the file in `modules/<TheModule>/metadata/studio.php` will be the only one checked. A file in `custom/modules/<TheModule>/metadata/studio.php` will have no effect.

7. Controllers

SuiteCRM follows the MVC (Model-View-Controller) pattern and as such has the concept of controllers. The controller is responsible for making changes to the Model as well as passing control to the view as appropriate. SuiteCRM has the concept of actions which are actions that will be taken by the controller. Let's take a look at a SuiteCRM URL:

Example 7.1: Example SuiteCRM URL

```
example.com/index.php?module=Accounts&action=index
```

In this (rather boring) example we see that the module is Accounts. This will determine which controller to use and then call the index action on that controller.

SuiteCRM will first look for the controller in `custom/module/<TheModule>/controller.php`. If this is not found then next `module/<TheModule>/controller.php` will be checked. Finally if neither of these controllers exist then the default controller will be used. The default controller can be found in `include/MVC/Controller/SugarController.php`.

Customising controllers

Ordinarily the default controller handles the request and delegates to the appropriate views etc. However custom controllers can be used to add or alter functionality. Let's look at adding a new action.

In the first instance we will have to add our custom controller. This will vary slightly depending on the nature of the module.

Custom module

In this case we can place the file directly into our module. You should create a new file (if it doesn't exist) at `modules/<TheModule>/controller.php`. The contents will look similar to:

Example 7.2: Creating a custom controller for a custom module

```php
<?php
if(!defined('sugarEntry') || !sugarEntry) die('Not A Valid Entry Point');
class <TheModule>Controller extends SugarController
{

}
```

Pre-existing modules

For pre-existing modules you should add the controller to
`custom/modules/<TheModule>/controller.php`.

The contents of this file will vary depending on whether you wish to extend the existing controller (if it exists) or create your own version completely. It is usually best to extend the existing controller since this will retain important logic. You should note the naming convention here. We name the class
`Custom<TheModule>Controller`.

Here we don't extend the existing controller or no such controller exists:

Example 7.3: Creating a custom controller for an existing module

```php
<?php
if(!defined('sugarEntry') || !sugarEntry) die('Not A Valid Entry Point');
class Custom<TheModule>Controller extends SugarController
{

}
```

Alternatively we extend the existing controller. Note that we are requiring the existing controller:

Example 7.4: Creating a custom controller for an existing module with an existing controller

```php
<?php
if(!defined('sugarEntry') || !sugarEntry) die('Not A Valid Entry Point');

require_once 'modules/<TheModule>/controller.php';

class Custom<TheModule>Controller extends <TheModule>Controller
{

}
```

Adding the action

Now we can add a new action to our controller. Actions are created as methods on the controller with the name `action_<actionName>`. For example, to create a new action called 'echo' we could create the following method in one of the controllers we have created above. This can then perform whatever logic that is needed. In our example we will log the REQUEST and simply redirect:

Example 7.5: Adding a custom controller action method

```php
public function action_echo(){
  $GLOBALS['log']->debug("Echo called with request: ".print_r($_REQUEST,1));
  SugarApplication::redirect('index.php');
}
```

Legacy Style

In previous versions of SugarCRM a new action was added by creating a file in either `modules/<TheModule>/<actionname>.php` or `custom/modules/<TheModule>/<actionname>.php`. Although this still works it is not recommended.

8. Entry Points

Entry points are simply a page which provides access to SuiteCRM. These can be used for a variety of purposes such as allowing an external form simple access to SuiteCRM or, as is the case with the stock Events module, allowing an event invite to be responded to by clicking a link in an email.

Creating an entry point

Let's create a simple entry point to display the time. First we define this entry point in a new file in:

Example 8.1: Entry point registry location

```
custom/Extension/application/Ext/EntryPointRegistry/
```

For our example we'll call our new file MyTimeEntryPoint.php

Example 8.2: Example entry point location

```
custom/Extension/application/Ext/EntryPointRegistry/MyTimeEntryPoint.php
```

In this file we will add a new entry to the `$entry_point_registry`. We supply the file that should be called. Here we are simply placing the file in custom if the entry point is related to a specific module it is usually a good idea to place this somewhere inside `custom/<TheModule>/`.

In addition we supply an "auth" parameter. If "auth" is true then anyone accessing the entry point will need to be logged into SuiteCRM.

Example 8.3: Adding an entry point entry

```php
<?php
        $entry_point_registry['MyTimeEntryPoint'] = array(
            'file' => 'custom/MyTimeEntryPoint.php',
            'auth' => true,
        );
```

Finally we add the actual logic itself inside custom/MyTimeEntryPoint.php:

Example 8.4: Example entry point that outputs the current time

```php
<?php
if(!defined('sugarEntry') || !sugarEntry) die('Not A Valid Entry Point');
$date = new DateTime();
echo $date->format('r');
```

After a Quick Repair and Rebuild we can access our entry point:

Example 8.5: Custom entry point URL

```
example.com/index.php?entryPoint=MyTimeEntryPoint
```

and we should see something similar to:

Example 8.6: MyTimeEntryPoint

```
Sun, 15 Mar 2015 13:03:03 +0000
```

Obviously this is a contrived example but any logic that can be performed elsewhere in SuiteCRM can be performed in an entry poiny (for example creating or editing SugarBeans).

9. Language Strings

Language strings provide an element of internationalisation to SuiteCRM. It allows specifying different strings to be used in different languages making it much easier to provide translations for modules and customisations. Even if you are only targeting a single language it is still worth using the language string functionality in SuiteCRM because it allows the simple changing of strings within SuiteCRM and it also allows users to customise the labels used in your customisations. There are three main types of language strings that we will cover here.

At the core, the language strings are a key value store. The keys are used throughout SuiteCRM and the values are loaded based on the current language.

Languages are handled in SuiteCRM by prefixing the file name with the IETF language code for the language that this file contains. Here are some examples of different language file names:

Example 9.1: Example language file names

```
# Core Accounts language file for en_us (United States English)
modules/Accounts/language/en_us.lang.php

# Core Cases language file for es_es (Spanish as spoken in Spain)
modules/Cases/language/es_es.lang.php

# Custom language file for de_de (German)
custom/Extension/application/Ext/Language/de_de.SomeCustomPackage.php
```

SuiteCRM will choose the language prefix to be used based on the language the user selected when logging in or the default language if none was selected. Generally when a language file is loaded the default language files and the `en_us` files will also be loaded. These files are then merged. This ensures that there will still be a definition if there are language keys in either `en_us` or the default language that don't have definitions in the current language. In essence the language "falls back" to the default language and `en_us` if there are missing keys.

Module Strings

Use

Module strings are strings associated with a particular module. These are usually, for example, field labels and panel name labels, but they may be used for anything that is specific to a single module.

Definition location

Module strings are defined in the `$mod_strings` array. This is initially defined in `modules/<TheModule>/language/<LanguageTag>.lang.php`, for example `modules/Accounts/language/en_us.lang.php`.

Customisation location

Customisations can be made to the module strings by adding a new file in `custom/Extension/modules/<TheModule>/Ext/Language/<LanguageTag>.<Name>.php` (`<Name>` in this case should be used to give it a descriptive name). An example is `custom/Extension/modules/Accounts/Ext/Language/en_us.MyLanguageFile.php`. See the Extensions section for more information on the Extensions folder.

Application Strings

Use

Application strings are used for language strings and labels that are not specific to a single module. Examples of these may include labels that will appear in the headers or footers, labels that appear on search buttons throughout SuiteCRM or labels for pagination controls.

Definition location

The application strings are defined in the `$app_strings` array. This is initially defined in `include/language/<LanguageTag>.lang.php`.

Customisation location

Customisations can be made to the application strings in two ways. Firstly you can edit the file `custom/include/language/<LanguageTag>.lang.php`. However to promote modularity it is recommended that you add a new file in the location `custom/Extension/application/Ext/Language/<LanguageTag>.<Name>.php`. For example `custom/Extension/application/Ext/Language/es_es.MyAppLanguageFile.php`. `<Name>` should be used to give the file a descriptive name. See the Extensions section for more information on the Extensions folder.

Application List Strings

Use

Application list strings are used to store the various dropdowns and lists used in SuiteCRM. Most of these are used as options for the various enum fields in SuiteCRM e.g the account type or the opportunity sales stage.

Definition location

The application list strings are defined in the `$app_list_strings` array. Similar to the `$app_strings` array this is initially defined in `include/language/en_us.lang.php`.

Customisation location

Customisations can be made to the application list strings in two ways. Firstly you can edit the file `custom/include/language/<LanguageTag>.lang.php`. However to promote modularity it is recommended that you add a new file in the location `custom/Extension/application/Ext/Language/<LanguageTag>.<Name>.php` (`<Name>` should be used to give the file a descriptive name). For example `custom/Extension/application/Ext/Language/es_es.MyAppListLanguageFile.php`. See the Extensions section for more information on the Extensions folder.

Why and when to customise

Generally language strings should be changed from within SuiteCRM using the studio tool. However there are times when it can be simpler to add or modify language strings as described in the previous section. If you are importing a large number of language strings or dropdown options it can be simpler to create a new file to add these values. Similarly if you are adding entirely new functionality, it is usually best to simply add these language strings as new values.

Usage

Language strings are used automatically throughout SuiteCRM. For example in metadata you can specify the language strings to display for fields. However in some cases you will want to access and use the language strings in custom code. There are several ways to do this.

Globals

The `$mod_strings`, `$app_strings` and `$app_list_strings` variables are all global and can be accessed as such. `$app_strings` and `$app_list_strings` will always be available. However `$mod_strings` will only contain the strings for the current module (see the next section for other ways of accessing `$mod_strings`).

Example 9.2: Accessing language strings globally

```
function someFunction(){
    global $mod_strings, $app_strings, $app_list_strings;
    /*
     * Grab the label LBL_NAME for the current module
     * In most modules this will be the label for the
     * name field of the module.
     */
    $modLabel = $mod_strings['LBL_NAME'];

    $appLabel = $app_strings['LBL_GENERATE_LETTER'];

    /*
     * Unlike the previous two examples $appListLabel will be an
     * array of the dropdowns keys to it's display labels.
     */
    $appListLabel = $app_list_strings['aos_quotes_type_dom'];

    //Here we just log out the strings
    $GLOBALS['log']->debug("The module label is $modLabel");
    $GLOBALS['log']->debug("The app label is $appLabel");
    $GLOBALS['log']->debug("The app list label is ".print_r($appListLabel,1));
}
```

Translate

As an alternative to using globals or, if you are in a different module than the language string you wish to retrieve you can use the `translate` method.

Example 9.3: `translate` method signature

```
translate(
        $string,
        $mod='',
        $selectedValue='')
```

$string
: The language string to be translated.

$mod
: The module this string should come from. Defaults to the current module if empty.

$selectedValue
: For dropdown strings. This will return the label for the key `$selectedValue`

Here is an example of the above in action. Note that we do not have to worry about whether the label is a Module string, an Application string or an Application list string, as all of these will be checked (in that order - the first matching value will be returned).

Example 9.4: Example `translate` method calls

```
function someFunction(){
  //Grab the label LBL_NAME for the current module
  $modLabel = translate('LBL_NAME');

  //Grab the label LBL_NAME for the AOS_Products module
  $productModLabel = translate('LBL_NAME','AOS_Products');

  $appLabel = translate('LBL_GENERATE_LETTER');

  /*
   * Return the label for the `Other` option of the `aos_quotes_type_dom`
   * We don't care about the module so this is left blank.
   */
  $appListLabel = translate('aos_quotes_type_dom','','Other');

  //Here we just log out the strings
  $GLOBALS['log']->debug("The module label is $modLabel");
  $GLOBALS['log']->debug("The module label for Products is $productModLabel");
  $GLOBALS['log']->debug("The app label is $appLabel");
  $GLOBALS['log']->debug("The app list label is ".print_r($appListLabel,1));
}
```

JavaScript

Finally, you may be using JavaScript (for example in a view), and wish to display a language string. For this you can use the `SUGAR.language.get` method, which is similar to the `translate` method in example 9.3.

Example 9.5: `SUGAR.language.get` method signature

```
SUGAR.language.get(
              module,
              str
);
```

module
: The module a language string will be returned for. You should supply `app_strings` or `app_list_strings` if the label you wish to retrieve is not a module string.

str
: The key you want to retrieve a label for.

Example 9.6: Example `SUGAR.language.get` method calls

```
function someFunction(){

  /*
   * Grab the label LBL_NAME for AOS_Products
   * Note that, unlike the translate function in example 9.3
   * the module name is required.
   */

  var modLabel = SUGAR.language.get('AOS_Products', 'LBL_NAME');

  /*
   * As mentioned above we explicitly need to pass if we are retrieving
   * an app_string or app_list_string
   */
  var appLabel = SUGAR.language.get('app_strings', 'LBL_GENERATE_LETTER');
  var appListLabel = SUGAR.language.get('app_list_strings',
                                        'aos_quotes_type_dom');

  //Here we just log out the strings
  console.log("The module label is "+modLabel);
  console.log("The app label is "+appLabel);
  console.log("The app list label is "+appListLabel);
}
```

10. Config

The config files

There are two main config files in SuiteCRM, both of which are in the root SuiteCRM folder. These are `config.php` and `config_override.php`. The definitions in here provide various configuration options for SuiteCRM. All the way from the details used to access the database to how many entries to show per page in the list view. Most of these options are accessible from the SuiteCRM administration page. However some are only definable in the config files.

config.php

This is the main SuiteCRM config file and includes important information like the database settings and the current SuiteCRM version.

Generally settings in this file wont be changed by hand. An exception to this is if SuiteCRM has been moved or migrated. In which case you may need to change the database settings and the site_url. Let's look at the database settings first:

Example 10.1: Database config definition

```
'dbconfig' =>
array (
  'db_host_name' => 'localhost',
  'db_host_instance' => 'SQLEXPRESS',
  'db_user_name' => 'dbuser',
  'db_password' => 'dbpass',
  'db_name' => 'dbname',
  'db_type' => 'mysql',
  'db_port' => '',
  'db_manager' => 'MysqliManager',
),
```

Here we can see this instance is setup to access a local MySQL instance using the username/password dbuser/dbpass and accessing the database named 'dbname'.

The site url settings are even simpler:

Example 10.2: Setting the site URL

```
  'site_url' => 'http://example.com/suitecrm',
```

The site url for the above is simply 'http://example.com/suitecrm' if we were moving this instance to, for example, suite.example.org, then we can simply place that value in the file.

These are generally the only two instances where you would directly change `config.php`. For other changes you would either make the change through SuiteCRM itself or you would use the `config_override.php` file.

config_override.php

`config_override.php` allows you to make config changes without risking breaking the main config file. This is achieved quite simply by adding, editing or removing items from the $sugar_config variable. The `config_override.php` file will be merged with the existing config allowing, as the name suggests, overriding the config. For example in config_override.php we can add our own, new, config item:

Example 10.3: Adding a custom config value

```
$sugar_config['enable_the_awesome'] = true;
```

or we can edit an existing config option in a very similar manner by simply overwriting it:

Example 10.4: Overwriting an existing config value

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

Using config options

We may want to access config options in custom code (or as detailed above if we have created our own config setting we may want to use that). We can easily get the config using the php global keyword:

Example 10.5: Accessing a config setting within SuiteCRM

```
function myCustomLogic(){
  //Get access to config
  global $sugar_config;
  //use the config values
  if(!empty($sugar_config['enable_the_awesome'])){
    doTheAwesome();
  }
}
```

11. Logging

Logging messages

Logging in SuiteCRM is achieved by accessing the log global. Accessing an instance of the logger is as simple as

Example 11.1: Accessing the log

```
$GLOBALS['log']
```

This can then be used to log a message. Each log level is available as a method. For example:

Example 11.2: Logging messages

```
$GLOBALS['log']->debug('This is a debug message');
$GLOBALS['log']->error('This is an error message');
```

This will produce the following output:

Example 11.3: Logging messages example output

```
Tue Apr 28 16:52:21 2015 [15006][1][DEBUG] This is a debug message
Tue Apr 28 16:52:21 2015 [15006][1][ERROR] This is an error message
```

Logging output

The logging output displays the following information by default:

Example 11.4: Logging messages example output

```
<Date> [<ProcessId>][<UserId>][<LogLevel>] <LogMessage>
```

`<Date>`
The date and time that the message was logged.

`<ProcessId>`
The PHP process id.

`<UserId>`
: The ID of the user that is logged into SuiteCRM.

`<LogLevel>`
: The log level for this log message.

`<LogMessage>`
: The contents of the log message.

Log levels

Depending on the level setting in admin some messages will not be added to the log e.g if your logger is set to `error` then you will only see log levels of `error` or higher (`error`, `fatal` and `security`).

The default log levels (in order of verbosity) are:

- `debug`
- `info`
- `warn`
- `deprecated`
- `error`
- `fatal`
- `security`

Generally on a production instance you will use the less verbose levels (probably `error` or `fatal`). However whilst you are developing you can use whatever level you prefer. I prefer the most verbose level - `debug`.

12. Logic Hooks

Intro

Logic hooks allow you to hook into various events in SuiteCRM to fire custom code. This can allow you to, for example, make a call to an external API, or to create a new record if certain events occur.

Types

Logic hooks can occur in three contexts. These contexts are Application Hooks, Module Hooks and User Hooks. These are detailed below.

Application Hooks

Application hooks are hooks which are fired in the application context (that is, they are not fired against a particular module). These hooks must be defined in the top level logic hook (i.e. `custom/modules/logic_hooks.php`).

after_entry_point
: Called after SuiteCRM has initialised but before any other processing is carried out.

after_ui_footer
: Called after the UI footer.

after_ui_frame
: Fired after the UI has been displayed but before the footer has been displayed.

server_round_trip
: Fired at the end of every page request.

User Hooks

User hooks are fired for certain login/logout actions. Similar to Application Hooks, these hooks must be defined in the top level logic hook (i.e. custom/modules/logic_hooks.php).

after_login
: Fired after a user logs in to SuiteCRM .

after_logout
Fired when a user logs out of SuiteCRM.

before_logout
Fired before a user logs out of SuiteCRM.

login_failed
Fired when a user attempts to login to SuiteCRM but the login fails.

Module Hooks

Module Hooks are called on various record actions for a specific module.

after_delete
Fired when a record is deleted.

after_relationship_add
Fired after a relationship is added between two records. Note that this may be called twice, once for each side of the relationship.

after_relationship_delete
Fired after a relationship between two records is deleted.

after_restore
Fired after a record is undeleted.

after_retrieve
Fired after a record is retrieved from the DB.

after_save
Fired after a record is saved. Note that due to some peculiarities some related modules may not be persisted to the database. The logic hook is fired within the SugarBean classes save method. Some implementing classes may save related beans after this method returns. A notable example of this is the saving of email addresses in Company modules.

before_delete
Fired before a record is deleted.

before_relationship_add
Fired before a relationship is added between two records. Note that this may be called twice, once for each side of the relationship.

before_relationship_delete
Fired before a relationship between two records is deleted. Note that this may be called twice, once for each side of the relationship.

before_restore
: Fired before a record is undeleted.

before_save
: Fired before a record is saved.

handle_exception
: Fired when an exception occurs in a record.

process_record
: Fired when a record is processed ready to be displayed in list views or dashlets.

Job Queue Hooks

Job queue hooks are fired for scheduler jobs. Similar to application hooks these hooks must be defined in the top level logic hook (i.e. `custom/modules/logic_hooks.php`).

job_failure
: Fired when a scheduled job either returns false to signify failure or throws an exception and it will not be retried. See the section on Scheduled Tasks.

job_failure_retry
: Fired when a scheduled job either returns false to signify failure or throws an exception but it will be retried. See the section on Scheduled Tasks.

Implementing

Depending on the Logic Hook type logic hooks are either placed into `custom/modules/Logic_Hooks.php` or `custom/modules/<TargetModule>/Logic_Hooks.php`.

Logic_Hooks.php

The logic hook file itself specifies which logic hooks to fire on this event. It looks something like this:

Example 12.1: Logic hook file

```php
<?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(
                              77,
                              'updateGeocodeInfo',
                              'custom/modules/Cases/CasesJjwg_MapsLogicHook.php',
                              'CasesJjwg_MapsLogicHook',
                              'updateGeocodeInfo');
$hook_array['before_save'][] = Array(
                              10,
                              'Save case updates',
                              'modules/AOP_Case_Updates/CaseUpdatesHook.php',
                              'CaseUpdatesHook',
                              'saveUpdate');
$hook_array['before_save'][] = Array(
                              11,
                              'Save case events',
                              'modules/AOP_Case_Events/CaseEventsHook.php',
                              'CaseEventsHook',
                              'saveUpdate');
$hook_array['before_save'][] = Array(
                              12,
                              'Case closure prep',
                              'modules/AOP_Case_Updates/CaseUpdatesHook.php',
                              'CaseUpdatesHook',
                              'closureNotifyPrep');
$hook_array['before_save'][] = Array(
                              1,
                              'Cases push feed',
                              'custom/modules/Cases/SugarFeeds/CaseFeed.php',
                              'CaseFeed',
                              'pushFeed');
$hook_array['after_save'] = Array();
$hook_array['after_save'][] = Array(
                              77,
                              'updateRelatedMeetingsGeocodeInfo',
                              'custom/modules/Cases/CasesJjwg_MapsLogicHook.php',
                              'CasesJjwg_MapsLogicHook',
                              'updateRelatedMeetingsGeocodeInfo');
$hook_array['after_save'][] = Array(
                              10,
                              'Send contact case closure email',
                              'modules/AOP_Case_Updates/CaseUpdatesHook.php',
                              'CaseUpdatesHook',
                              'closureNotify');
$hook_array['after_relationship_add'] = Array();
$hook_array['after_relationship_add'][] = Array(
                              77,
                              'addRelationship',
                              'custom/modules/Cases/CasesJjwg_MapsLogicHook.php',
                              'CasesJjwg_MapsLogicHook',
                              'addRelationship');
$hook_array['after_relationship_add'][] = Array(
                              9,
                              'Assign account',
                              'modules/AOP_Case_Updates/CaseUpdatesHook.php',
                              'CaseUpdatesHook',
                              'assignAccount');
$hook_array['after_relationship_add'][] = Array(
                              10,
                              'Send contact case email',
                              'modules/AOP_Case_Updates/CaseUpdatesHook.php',
                              'CaseUpdatesHook',
                              'creationNotify');
$hook_array['after_relationship_delete'] = Array();
$hook_array['after_relationship_delete'][] = Array(
                              77,
                              'deleteRelationship',
                              'custom/modules/Cases/CasesJjwg_MapsLogicHook.php',
                              'CasesJjwg_MapsLogicHook',
                              'deleteRelationship');
```

Let's go through each part of the file.

```
$hook_version = 1;
```

This sets the hook version that we are using. Currently there is only one version so this line is unused.

```
$hook_array = Array();
```

Here we set up an empty array for our Logic Hooks. This should always be called $hook_array.

```
$hook_array['before_save'] = Array();
```

Here we are going to be adding some before_save hooks so we add an empty array for that key.

```
$hook_array['before_save'][] = Array(
                              77,
                              'updateGeocodeInfo',
                              'custom/modules/Cases/CasesJjwg_MapsLogicHook.php',
                              'CasesJjwg_MapsLogicHook',
                              'updateGeocodeInfo');
```

Finally we reach an interesting line. This adds a new logic hook to the before_save hooks. This array contains 5 entries which define this hook. These are:

Sort order

The first argument (77) is the sort order for this hook. The logic hook array is sorted by this value. If you wish for a hook to fire earlier you should use a lower number. If you wish for a hook to be fired later you should use a higher number. The numbers themselves are arbitrary.

Hook label

The second argument ('updateGeocodeInfo') is simply a label for the logic hook. This should be something short but descriptive.

Hook file

The third argument is where the actual class for this hook is. In this case it is in a file called `custom/modules/Cases/CasesJjwg_MapsLogicHook.php`. Generally you will want the files to be somewhere in custom and it is usual to have them in `custom/modules/<TheModule>/<SomeDescriptiveName>.php` or `custom/modules/<SomeDescriptiveName>.php` for Logic Hooks not targeting a specific module. However the files can be placed anywhere.

Hook class

The fourth argument is the class name for the Logic Hook class. In this case `CasesJjwg_MapsLogicHook`. It is usual for the class name to match the file name but this is not required.

Hook method

The fifth, and final, argument is the method that will be called on the class. In this case `update-GeocodeInfo`.

Adding your own logic hooks

When adding logic hooks you should make full use of the Extensions framework (see the section on Extensions). This involves creating a file in
`custom/Extension/application/Ext/LogicHooks/` for application hooks and
`custom/Extension/modules/<TheModule>/Ext/LogicHooks/` for module specific hooks. These files can then add to/alter the `$hook_array` as appropriate.

After adding a new logic hook it is necessary to perform a quick repair and rebuild in the admin menu for this to be picked up.

Logic Hook function

The logic hook function itself will vary slightly based on the logic hook type. For module hooks it will appear similar to:

Example 12.2: Example logic hook method

```
class SomeClass
{
    function someMethod($bean, $event, $arguments)
    {
      //Custom Logic
    }
}
```

Application logic hooks omit the $bean argument:

Example 12.3: Example logic hook method for application hooks

```
    class SomeClass
    {
        function someMethod($event, $arguments)
        {
          //Custom Logic
        }
    }
```

$bean (`SugarBean`)

The $bean argument passed to your logic hook is usually the bean that the logic hook is being performed on. For User Logic Hooks this will be the current User object. For module logic hooks (such as `before_save`) this will be the record that is being saved. For job queue logic hooks this will be the SchedulersJob bean. Note that for Application Logic Hook this argument is not present.

$event (`string`)

The $event argument contains the logic hook event e.g `process_record`, `before_save`, `after_delete` etc.

$arguments (`array`)

The $arguments argument contains any additional details of the logic hook event. I.e. in the case of before_relationship_add this will contain details of the related modules.

Tips

Triggering extra logic hooks

If you are performing certain actions that may trigger another logic hook (such as saving a bean) then you need to be aware that this will trigger the logic hooks associated with that bean and action. This can be troublesome if this causes a logic hook loop of saves causing further saves. One way around this is to simply be careful of the hooks that you may trigger. If doing so is unavoidable you can usually set an appropriate flag on the bean and then check for that flag in subsequent hooks.

Think of the user

Most logic hooks will cause additional code which can degrade the users experience. If you have long running code in the after_save the user will need to wait for that code to run. This can be avoided by either ensuring the code runs quickly or by using the Job Queue (see the Job Queue chapter for more information).

13. Scheduled Tasks

Intro

Scheduled tasks are performed in SuiteCRM by the scheduler module. Jobs are placed into the queue either through the defined scheduled tasks or, for one off tasks, by code creating job objects. Note that both scheduled tasks and using the job queue requires that you have the schedulers set up. This will vary depending on your system but usually requires adding an entry either to Cron (for Linux systems) or to the windows scheduled tasks (for windows). Opening the scheduled tasks page within SuiteCRM will let you know the format for the entry.

Scheduler

Scheduled tasks allow SuiteCRM to perform recurring tasks. Examples of these which ship with SuiteCRM include checking for incoming mail, sending email reminder notifications and indexing the full text search. What if you want to create your own tasks?

SuiteCRM lets you define your own Scheduler. We do this by creating a file in `custom/Extension/modules/Schedulers/Ext/ScheduledTasks/`. You can give this file a name of your choice but it is more helpful to give it a descriptive name. Let's create a simple file named `custom/Extension/modules/Schedulers/Ext/ScheduledTasks/CleanMeetingsScheduler.php`. This will add a new job to the job strings and a new method that the scheduler will call:

Example 13.1: Example Clean Meetings Scheduler

```php
<?php
/*
 * We add the method name to the $job_strings array.
 * This is the method that jobs for this scheduler will call.
 */
$job_strings[] = 'cleanMeetingsScheduler';

/**
 * Example scheduled job to change any 'Planned' meetings older than a month
 * to 'Not Held'.
 * @return bool
 */
function cleanMeetingsScheduler(){
  //Get the cutoff date for which meetings will be considered
```

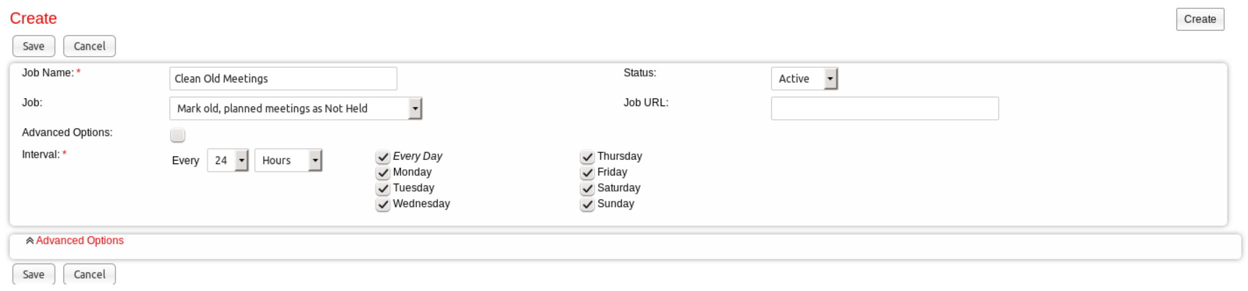
```php
        $cutOff = new DateTime('now - 1 month');
        $cutOff = $cutOff->format('Y-m-d H:i:s');

        //Get an instance of the meetings bean for querying
  //see the Working With Beans chapter.
  $bean = BeanFactory::getBean('Meetings');

  //Get the list of meetings older than the cutoff that are marked as Planned
  $query = "meetings.date_start < '$cutOff' AND meetings.status = 'Planned'";
  $meetings = $bean->get_full_list('',$query);

  foreach($meetings as $meeting){
    //Mark each meeting as Not Held
    $meeting->status = 'Not Held';
    //Save the meeting changes
    $meeting->save();
  }
  //Signify we have successfully ran
  return true;
}
```

We also make sure that we add a language file in `custom/Extension/modules/Schedulers/Ext/Language/en_us.cleanMeetingsScheduler.php` again, the name of the file doesn't matter but it is helpful to use something descriptive. This will define the language string for our job so the user sees a nice label. See the section on language strings for more information. The key for the mod strings will be LBL_UPPERMETHODNAME. In our case our method name is `cleanMeetingsScheduler` so our language label key will be `LBL_CLEANMEETINGSSCHEDULER`.

Example 13.2: Example Language string for Clean Meetings Scheduler

```php
<?php
//We add the mod string for our method here.
$mod_strings['LBL_CLEANMEETINGSSCHEDULER'] = 'Mark old, planned meetings as Not \
Held';
```

If we perform a repair and rebuild our method will now be packaged up into the scheduler ext file (see the Extensions section for more information on this process) and will be available in the schedulers page. Note that for any changes to the scheduler method you will need to perform another quick repair and rebuild - even in developer mode. We can now create a new scheduler to call our new method:


Create

Job Name: * Clean Old Meetings
Status: Active
Job: Mark old, planned meetings as Not Held
Job URL:
Advanced Options:
Interval: * Every 24 Hours
✓ Every Day ✓ Monday ✓ Tuesday ✓ Wednesday ✓ Thursday ✓ Friday ✓ Saturday ✓ Sunday

Creating a scheduler that uses our new method

This will now behave just like any other scheduler and we can have this run as often (or as rarely) as we would like. Take care here. The default frequency is every one minute. If your task is heavy duty or long running this may not be what you would prefer. Here we settle for once a day.

Job Queue

Sometimes you will require code to perform a long running task but you do not need the user to wait for this to be completed. A good example of this is sending an email in a logic hook (see the Logic Hooks chapter for information on these). Assuming we have the following logic hook:

Example 13.3: Example Email sending Logic Hook

```
class SomeClass
{
    function SomeMethod($bean, $event, $arguments)
    {

        //Perform some setup of the email class
        require_once "include/SugarPHPMailer.php";
        $mailer=new SugarPHPMailer();
        $admin = new Administration();
        $admin->retrieveSettings();
        $mailer->prepForOutbound();
        $mailer->setMailerForSystem();
        $admin = new Administration();
        $admin->retrieveSettings();
        $admin->settings['notify_fromname']
        $mailer->From     = $admin->settings['notify_fromaddress']
        $mailer->FromName = $emailSettings['from_name'];
        $mailer->IsHTML(true);

        //Add message and recipient.
        //We could go all out here and load and populate an email template
        //or get the email address from the bean
        $mailer->Subject = 'My Email Notification! '.$bean->name;
        $mailer->Body = $event. ' fired for bean '.$bean->name;
        $mailer->AddAddress('Jim@example.com');
        return $mailer->Send();
    }
}
```

This will work fine. However you do not want the user to have to wait for the email to be sent out as this can cause the UI to feel sluggish. Instead you can create a Job and place it into the job queue and this will be picked by the scheduler. Let's look at an example of how you would do this.

First we want our Logic Hook class to create the scheduled job:

Example 13.4: Example Scheduled Job Creation

```
class SomeClass
{
    function SomeMethod($bean, $event, $arguments)
    {
      require_once 'include/SugarQueue/SugarJobQueue.php';
      $scheduledJob = new SchedulersJob();

      //Give it a useful name
      $scheduledJob->name = "Email job for {$bean->module_name} {$bean->id}";

      //Jobs need an assigned user in order to run. You can use the id
      //of the current user if you wish, grab the assigned user from the
      //current bean or anything you like.
      //Here we use the default admin user id for simplicity
      $scheduledJob->assigned_user_id = '1';

      //Pass the information that our Email job will need
      $scheduledJob->data = json_encode(array(
                                              'id' => $bean->id,
                                              'module' => $bean->module_name)
                                          );

      //Tell the scheduler what class to use
      $scheduledJob->target = "class::BeanEmailJob";

      $queue = new SugarJobQueue();
      $queue->submitJob($scheduledJob);
    }
}
```

Next we create the BeanEmailJob class. This is placed into the `custom/Extensions/modules/Schedulers/Ext/ScheduledTasks/` directory with the same name as the class. So in our example we will have:
`custom/Extensions/modules/Schedulers/Ext/ScheduledTasks/BeanEmailJob.php`

Example 13.5: Example Scheduler job

```
class BeanEmailJob implements RunnableSchedulerJob
{
  public function run($arguments)
  {

    //Only different part of the email code.
    //We grab the bean using the supplied arguments.
    $arguments = json_decode($arguments,1);
    $bean = BeanFactory::getBean($arguments['module'],$arguments['id']);

    //Perform some setup of the email class
    require_once "include/SugarPHPMailer.php";
    $mailer=new SugarPHPMailer();
    $admin = new Administration();
    $admin->retrieveSettings();
    $mailer->prepForOutbound();
    $mailer->setMailerForSystem();
    $admin = new Administration();
    $admin->retrieveSettings();
    $mailer->From     = $admin->settings['notify_fromaddress'];
    $mailer->FromName = $emailSettings['from_name'];
    $mailer->IsHTML(true);

    //Add message and recipient.
    //We could go all out here and load and populate an email template
    //or get the email address from the bean
    $mailer->Subject = 'My Email Notification! '.$bean->name;
    $mailer->Body = $event. ' fired for bean '.$bean->name;
    $mailer->AddAddress('Jim@example.com');
    return $mailer->Send();
  }
  public function setJob(SchedulersJob $job)
  {
    $this->job = $job;
  }
}
```

Now whenever a user triggers the hook it will be much quicker since we are simply persisting a little info to the database. The scheduler will run this in the background.

Retries

Occasionally you may have scheduled jobs which could fail intermittently. Perhaps you have a job which calls an external API. If the API is unavailable it would be unfortunate if the job failed and was never retried. Fortunately the SchedulersJob class has two properties which govern how retries are handled. These are `requeue` and `retry_count`.

`requeue`
Signifies that this job is eligible for retries.

`retry_count`
Signifies how many retries remain for this job. If the job fails this value will be decremented.

We can revisit our previous example and add two retries:

Example 13.6: Setting the retry count on a scheduled job

```
      $scheduledJob = new SchedulersJob();

      //Give it a useful name
      $scheduledJob->name = "Email job for {$bean->module_name} {$bean->id}";

      //Jobs need an assigned user in order to run. You can use the id
      //of the current user if you wish, grab the assigned user from the
      //current bean or anything you like.
      //Here we use the default admin user id for simplicity
      $scheduledJob->assigned_user_id = '1';

      //Pass the information that our Email job will need
      $scheduledJob->data = json_encode(array(
                                              'id' => $bean->id,
                                              'module' => $bean->module_name)
                                          );

      //Tell the scheduler what class to use
```

```
    $scheduledJob->target = "class::BeanEmailJob";

    //Mark this job for 2 retries.
    $scheduledJob->requeue = true;
    $scheduledJob->retry = 2;
```

See the section on logic hooks for more information on how job failures can be handled.

Debugging

With Scheduled tasks and jobs running in the background it can sometimes be difficult to determine what is going on when things go wrong. If you are debugging a scheduled task the the scheduled task page is a good place to start. For both scheduled tasks and job queue tasks you can also check the job_queue table. For example, in MySQL we can check the last five scheduled jobs:

Example 13.7: Example MySQL query for listing jobs

```
SELECT * FROM job_queue ORDER BY date_entered DESC LIMIT 5
```

This will give us information on the last five jobs. Alternatively we can check on specific jobs:

Example 13.8: Example MySQL query for listing BeanEmailJobs

```
SELECT * FROM job_queue WHERE target = 'class::BeanEmailJob'
```

In either case this will give details for the job(s):

Example 13.9: Example MySQL list of jobs

```
*************************** 1. row ***************************
assigned_user_id: 1
              id: 6cdf13d5-55e9-946e-9c98-55044c5cecee
            name: Email job for Accounts 103c4c9b-336f-0e87-782e-5501defb5900
         deleted: 0
    date_entered: 2015-03-14 14:58:15
   date_modified: 2015-03-14 14:58:25
    scheduler_id:
    execute_time: 2015-03-14 14:58:00
          status: done
      resolution: success
         message: NULL
```

```
          target: class::BeanEmailJob
            data: {"id":"103c4c9b-336f-0e87-782e-5501defb5900","module":"Account\
s"}
         requeue: 0
     retry_count: NULL
   failure_count: NULL
       job_delay: 0
          client: CRON3b06401793b3975cd00c0447c071ef9a:7781
percent_complete: NULL
1 row in set (0.00 sec)
```

Here we can check the status, resolution and message fields. If the status is `queued` then either the scheduler has not yet run or it isn't running. Double check your Cron settings if this is the case.

It may be the case that the job has ran but failed for some reason. In this case you will receive a message telling you to check the logs. Checking the logs usually provides enough information, particularly if you have made judicious use of logging (see the chapter on logging) in your job.

It is possible that the job is failing outright, in which case your logging may not receive output before the scheduler exits. In this case you can usually check your PHP logs.

As a last resort you can manually run the scheduler from the SuiteCRM directory using:

Example 13.10: Running the scheduler manually

```
php -f cron.php
```

Using this in addition to outputting any useful information should track down even the oddest of bugs.

14. Extension Framework

Introduction

The extension framework provides a means to modify various application data inside SuiteCRM. For example it provides a way to add or modify vardefs, scheduled tasks, language strings and more. In general a folder is provided in `custom/Extension` (the exact path depends on the extension). This folder is then scanned for files which will be consolidated into a single ext file which SuiteCRM will then read and use. In this way it is possible for developers to add a new file to affect the behaviour of SuiteCRM rather than altering existing files. This makes the changes more modular and allows the easy addition or removal of changes. Additionally, because these files are all consolidated it means that there is no affect on performance of checking a (possibly large) number of files. This is only done when performing a repair and rebuild in the admin menu.

Standard Extensions

List of standard SuiteCRM extensions

Extension Directory	Compiled file	Module	Description
ActionViewMap	action_view_map.ext.php		Used to map actions for a module to a specified view.
ActionFileMap	action_file_map.ext.php		Used to map actions for a module to a specified file.
ActionReMap	action_remap.ext.php		Used to map actions for a module to existing actions.
Administration	administration.ext.php	Administration	Used to add new sections to the administration panel.
EntryPointRegistry	entry_point_registry.ext.php	application	Used to add new entry points to SuiteCRM. See the chapter on Entry Points.
Extensions	extensions.ext.php	application	Used to add new extension types.
FileAccessControlMap	file_access_control_-map.ext.php		Used to add, update or delete entries in the access control lists for files.

List of standard SuiteCRM extensions

Extension Directory	Compiled file	Module	Description
Language	N/A[1]		Used to add, update or delete language strings for both modules and app strings. See the chapter on Language Strings.
Layoutdefs	layoutdefs.ext.php		Used to add, update or delete subpanel definitions for a module.
GlobalLinks	links.ext.php	application	Used to add, update or delete global links (the list of links that appear in the top right of the SuiteCRM UI).
LogicHooks	logichooks.ext.php		Used to add, update or delete logic hooks. See the chapter on Logic Hooks.
Include	modules.ext.php	application	Used to register new beans and modules.
Menus	menu.ext.php		Used to add, update or delete the menu links for each module.
ScheduledTasks	scheduledtasks.ext.php	Schedulers	Used to add new scheduled tasks. See the chapter on Scheduled Tasks.
UserPage	userpage.ext.php	Users	Unused
Utils	custom_utils.ext.php	application	Used to add new utility methods.
Vardefs	vardefs.ext.php		Used to add, update or delete vardefs for a module. See the section on Vardefs.
JSGroupings	jsgroups.ext.php		Used to add, update or delete JavaScript groupings.
Actions	actions.ext.php	AOW_Actions	Used to add new WorkFlow actions.

Custom Extensions

Interestingly the extension framework can be used to add new extensions. This allows you to create customisations that are easily customised by others (in a similar manner to, for example, how vardefs can be added - see the chapter on Vardefs).

To create a custom extension you simply add a new file in `custom/Extension/application/Ext/Extensions`. This can be given a name of your choosing. Our

[1]The language extensions are treated specially and, as such, aren't compiled to a single file.

example will use
`custom/Extension/application/Ext/Extensions/SportsList.php` and will look like:

Example 14.1: Adding an entry point entry

```php
<?php
$extensions["sports_list"] =  array(
                "section" => "sports_list",
                "extdir" => "SportsList",
                "file" => 'sportslist.ext.php',
                "module" => "");
```

Now when a Quick Repair and rebuild is run any files in
`custom/Extension/application/Ext/SportsList/` will be consolidated into
`custom/application/Ext/SportsList/sportslist.ext.php`. On it's own this file will not do anything but you are now able to write custom code that checks the consolidated file rather than having to worry about searching for customisations.

15. Module Installer

As detailed in the other chapters of this book there are many ways to customise SuiteCRM. The module installer allows you to package these changes and install them onto other SuiteCRM instances. This is achieved by creating a package.

At the minimum a package is a zip file that contains a `manifest.php` file in it's root. The manifest file is responsible for providing information about the installer as well as providing information on how to install the package.

manifest.php

The `manifest.php` file contains the definition of three arrays. Let's look at each of these arrays in turn. See Appendix A for the full sample `manifest.php` file.

Within path in the manifest file you can use `<basepath>` to refer to the base directory of the installer. For example `<basepath>/Foo.txt` will refer to the `Foo.txt` file in the root of the installer package.

`$manifest`

The `$manifest` array provides information on the package itself such as it's name, readme etc. (it also defines the `copy` array for `patch` packages). A sample definition of the manifest array will appear something like this:

Example 15.1: Example `$manifest` array definition

```php
$manifest = array(
  'name' => 'My First Package',
  'description' => 'This is a simple package example manifest file',
  'version' => '1.5',
  'author' => 'Jim Mackin',
  'readme' => 'readme.txt',
  'acceptable_sugar_flavors' => array('CE'),
  'acceptable_sugar_versions' => array(
    'exact_matches' => array(),
    'regex_matches' => array('6\\.5\\.[0-9]$'),
  ),
  'copy_files' => array (
    'from_dir' => '<basepath>/custom/',
    'to_dir' => 'custom',
    'force_copy' => array (),
  ),
  'dependencies' => array(
    array(
      'id_name' => 'example_dependency_package',
      'version' => '2.4',
    ),
  ),
);
```

`name` The name of the package. This is how the package will appear to the user during installation and in the Module Loader package list. The package name is required.

`description`
A brief description of the package.

`version`
The version of this package. This can be any string but is usually a traditional version number (such as `3.1.4`).

`author`
The author of the package.

`readme`
A brief readme string. Note that if a `README.txt` is found in the root of the package this will be used instead.

`acceptable_sugar_flavors`
A remnant of the SugarCRM packages. This should always be an array with (at least) a `CE` entry. If you would like the installer to target both SuiteCRM and SugarCRM editions then this can contain one of the other SugarCRM flavours (`PRO`, `CORP` , `ULT` or `ENT`).

`acceptable_sugar_versions`
An array detailing the matching SugarCRM versions. Note that the SugarCRM version is distinct from the SuiteCRM version. This array has two keys. `exact_matches` is simply an array of the allowed versions. `regex_matches` allows specifying regexes to match versions. For SuiteCRM you only need to worry about supporting the `6.5.*` versions which can be matched with the regex `6\\.5\\.[0-9]$`. At the time of writing the current SugarCRM version for SuiteCRM is `6.5.20`.

`copy_files`
: This is only used for `patch` installers and will copy files in the `from_dir` key to those in the `to_dir` key. Finally the `force_copy` key can be used to specify files that should be forcibly copied over.

`dependencies`
: An array of other packages that are relied on by this package. Each entry is an array with `id_name` - the id of the package and `version` - the required version of the package.

`icon` The path (within the installer) to an icon to be displayed during installation.

`is_uninstallable`
: Whether or not uninstalls should be allowed.

`published_date`
: The date that the package was published. There is no fixed format for the date, it is simply a string.

`key` Specifies a key to ensure that modules do not clash. This will prefix the installed modules and tables with `key`. This is used by the module builder when creating packages but can be specified if you wish.

`remove_tables`
: A string specifying whether module tables should be removed when uninstalling this package. Accepted values are `true`, `false` and `prompt`. The default is `true`.

`type` The type of the installer, one of `langpack`, `module`, `patch` or `theme`. See the types section.

`$install_defs`

Provides information on how the package is to be installed, which files go where and any additional information such as logic hooks, custom fields etc.

`id`

A unique identifier for the module.

`connectors`

An array of connectors to be installed. Each entry is an array with the following keys:

Key	Description
`name`	The name of the connector.
`connector`	The directory to copy the connector files from.
`formatter`	The directory to copy the connector formatter files from.

`copy`

An array of files and directories to be copied on install. Each entry is an array with the following keys:

Key	Description
`from`	The source file/directory in the package.
`to`	The destination file/directory.

In general if a file can be handled by one of the other keys then that key should be used. For example new admin entries should be copied using the `administration` key rather than using the `copy` key.

`dashlets`

An array of dashlets to be installed. Each entry is an array with the following keys:

Key	Description
`name`	The name of the new dashlet.
`from`	The path in the install package from which the dashlet files will be copied.

`language`

An array of language files to be installed. Each entry is an array with the following keys:

Key	Description
`from`	The location of the language file inside the package.
`to_module`	The module this language file is intended for (or 'application' for application language strings).
`language`	The language that this file is for (i.e. en_us or es_es).

See the chapter on Language Strings for more information.

`layoutdefs`

An array of layoutdef files which are used to add, remove or edit subpanels. Each entry is an array with the following keys:

Key	Description
`from`	The path in the package to the file to be installed.
`to_module`	The module that this file will be installed to.

`vardefs`

An array of the vardefs to be added to specific modules. Each entry is an array with the following keys:

Key	**Description**
`from`	The location of the vardef file in the package.
`to_module`	The destination module.

Generally you should install custom fields using the `custom_fields` key. However this key can be used to alter existing fields or add more complex fields.

`menu`

An array of menus to be installed. Each entry is an array with the following keys:

Key	**Description**
`from`	The location of the menu file in the package.
`to_module`	The destination module for this menu.

`beans`

An array of beans to be installed. Each entry is an array with the following keys:

Key	**Description**
`module`	The name of the module.
`class`	The name of the bean class.
`path`	The path (within the package) to the bean file.
`tab`	Whether or not a tab should be added for this module.

`relationships`

An array detailing any new relationships added (in particular relationships where one side is an existing module). Each entry is an array with the following keys:

Key	**Description**
`module`	The module that this relationship will be attached to.
`meta_data`	The location of the metadata file for this relationship.

`custom_fields`

An array of new custom fields to be installed (See the Vardefs chapter for more information on this). Each entry is an array with the following keys:

Key	Description
`name`	The name of the new custom field.
`label`	The key for the language string which will act as the label for this custom field.
`type`	The type of this custom field.
`max_size`	For string field types, the maximum number of characters.
`require_option`	Whether or not the field is required.
`default_value`	The default value of this field.
`ext1`	Extended field information. Different field types will use this value differently. For example Enum fields will store the key for the options in this field, decimal and float fields will store the precision.
`ext2`	Extended field information. Different field types will use this value differently. For example, dynamic dropdowns will store the parent dropdown, text areas will store the number of rows.
`ext3`	Extended field information. Different field types will use this value differently. For example, text areas will store the number of columns.
`ext4`	Extended field information. Different field types will use this value differently. For HTML field types this will store the HTML.
`audited`	Whether or not changes to this field should be audited.
`module`	Used to specify the module where the custom field will be added.

`logic_hooks`

An array of logic hooks to be installed. See the Logic Hooks chapter for more information. Each entry is an array with the following keys:

Key	Description
`module`	The module to where this logic hook should be installed. Leaving this empty will install into the top level logic hook.
`hook`	The logic hook type (i.e. `after_save`, `after_login`, etc.).
`order`	The sort order for this logic hook.
`description`	A description of the hook.
`file`	The file containing the class for this logic hook, relative to the SuiteCRM root.
`class`	The class that contains the logic hook function that should be called by this hook.
`function`	The function to be invoked when this hook is triggered.

`image_dir`

A path to a directory of images to be included in the install.

`schedulers`

An array of schedulers to be installed. Each entry is an array with a single key:

Key	Description
`from`	The file containing the new scheduled task.

`administration`

An array of admin panels to be installed. Each entry is an array with a single key:

Key	Description
`from`	The file containing the new admin panel definition.

`pre_execute`

Defines an array of files to be executed before the package is installed. Each entry is a path to a file within the package. Any output will be displayed to the user in the install log.

`post_execute`

Defines an array of files to be executed after the package is installed. Each entry is a path to a file within the package. Any output will be displayed to the user in the install log.

`pre_uninstall`

Defines an array of files to be executed before the package is uninstalled. Each entry is a path to a file within the package. Any output will be displayed to the user in the uninstall log.

`post_uninstall`

Defines an array of files to be executed after the package is uninstalled. Each entry is a path to a file within the package. Any output will be displayed to the user in the uninstall log.

`$upgrade_manifest`

Provides a means of upgrading an already installed package by providing different `install_defs`.

Types

Type	Description
langpack	A language installer. This will add an entry to the language dropdown.
module	A module installer. Will install new modules and/or functionality.
patch	A patch installer. This is used to upgrade SuiteCRM.
theme	A theme installer. This will add a new option to the themes.

Other files

`README.txt`
: Contains the readme for this package. If `README.txt` and a readme entry in the `manifest.php` is defined then this file will be used.

`LICENSE.txt`
: Provides information on the license for this package.

`scripts/pre_install.php`
: A PHP script which defines a method `pre_install()`. This method will be called before the package is installed. Any output will be displayed to the user in the install log.

`scripts/post_install.php`
: A PHP script which defines a method `post_install()`. This method will be called after the package is installed.

`scripts/pre_uninstall.php`
: A PHP script which defines a method `pre_uninstall()`. This method will be called before the package is uninstalled.

`scripts/post_uninstall.php`
: A PHP script which defines a method `post_uninstall()`. This method will be called after the package is uninstalled.

16. API

The SuiteCRM API allows third party code to access and edit SuiteCRM data and functionality.

Using the API

SuiteCRM has both a REST and a SOAP API. Which API you want to use will largely come down to personal preference and the support for SOAP/REST libraries in whichever language you will be using.

Both APIs will require a username and password. It is usual to create a user specifically for the API.

SOAP

The WSDL for the SOAP API can be found at:

Example 16.1: SOAP API WSDL Location

```
example.com/suitecrm/service/v4_1/soap.php?wsdl
```

Where `example.com/suitecrm` is the address of your SuiteCRM instance. `v4_1` is the version of the API and can be changed, `v4_1` is the latest version at the time of writing.

SOAP Example

The following PHP example uses the built in SoapClient class.

Example 16.2: Accessing the SOAP API

```php
<?php
//Create a new SoapClient
$wsdlURL = "http://example.com/suitecrm/service/v4_1/soap.php?wsdl";
$client = new SoapClient($wsdlURL);

//Login to the API and get the session id
$userAuth = array(
        'user_name' => '<suitecrmuser>',
        'password' => md5('<suitecrmpassword>'),
);
```

```php
$appName = 'My SuiteCRM SOAP Client';
$nameValueList = array();
$loginResults = $client->login($userAuth, $appName, $nameValueList);

//Get a list of at most 10 accounts with a billing address in Ohio. Along with
//The first and last names of any contacts in that Account.
$results = $client->get_entry_list(
        //Session id - retrieved from login call
        $loginResults->id,
        //Module to get_entry_list for
        'Accounts',
        //Filter query - Added to the SQL where clause
        "accounts.billing_address_city = 'Ohio'",
        //Order by - unused
        '',
        //Start with the first record
        0,
        //Return the id and name fields
        array('id','name'),
        //Link to the "contacts" relationship and retrieve the
        //First and last names.
        array(
                array(
                        'name' => 'contacts',
                        'value' => array(
                                'first_name',
                                'last_name',
                        ),
                ),
        ),
        //Show 10 max results
        10,
        //Do not show deleted
        0
);
print_r($results);
```

REST

The SuiteCRM REST API can be found at:

Example 16.3: REST API Endpoint Location

```
example.com/suitecrm/service/v4_1/rest.php
```

Where example.com/suitecrm is the address of your SuiteCRM instance. `v4_1` is the version of the API and can be changed, `v4_1` is the latest version at the time of writing.

The SuiteCRM REST API is not a true REST API - all calls are performed as POSTs and all calls are to the base URL with the method passed in as a post argument.

The arguments to the REST API calls are:

method
The method which will be called, i.e. `login` or `get_entry_list`. See Appendix B for a list of API methods.

input_type
The input type of the rest_data. This is usually `JSON` but can also be `Serialize`.

response_type
How the response will be encoded. This is usually `JSON` but can also be `Serialize`.

rest_data
Any other arguments that are required by this method. This is passed as an encoded array. The encoding is determined by input_type.

Note that, for REST requests it is the order of the arguments that matter in `rest_data` and not the name.

Examples

Example 16.4: Accessing the REST API

```php
<?php

$url = "http://example.com/suitecrm/service/v4_1/rest.php";

function restRequest($method, $arguments){
        global $url;
        $curl = curl_init($url);
        curl_setopt($curl, CURLOPT_RETURNTRANSFER, 1);
        $post = array(
                        "method" => $method,
                        "input_type" => "JSON",
                        "response_type" => "JSON",
                        "rest_data" => json_encode($arguments),
        );

        curl_setopt($curl, CURLOPT_POSTFIELDS, $post);

        $result = curl_exec($curl);
        curl_close($curl);
        return json_decode($result,1);
}


$userAuth = array(
        'user_name' => 'suitecrmuser',
        'password' => md5('suitecrmpassword'),
);
$appName = 'My SuiteCRM REST Client';
$nameValueList = array();

$args = array(
            'user_auth' => $userAuth,
            'application_name' => $appName,
            'name_value_list' => $nameValueList);

$result = restRequest('login',$args);
$sessId = $result['id'];

$entryArgs = array(
  //Session id - retrieved from login call
        'session' => $sessId,
  //Module to get_entry_list for
        'module_name' => 'Accounts',
  //Filter query - Added to the SQL where clause,
        'query' => "accounts.billing_address_city = 'Ohio'",
  //Order by - unused
        'order_by' => '',
  //Start with the first record
        'offset' => 0,
  //Return the id and name fields
        'select_fields' => array('id','name',),
        //Link to the "contacts" relationship and retrieve the
        //First and last names.
        'link_name_to_fields_array' => array(
                        array(
                                        'name' => 'contacts',
                                        'value' => array(
                                                        'first_name',
                                                        'last_name',
                                        ),
                        ),
        ),
  //Show 10 max results
        'max_results' => 10,
  //Do not show deleted
        'deleted' => 0,
);
$result = restRequest('get_entry_list',$entryArgs);

print_r($result);
```

For a full list of API methods and their arguments see Appendix B.

Adding custom API methods

Sometimes the existing API methods are not sufficient or using them for a task would be overly complex. SuiteCRM allows the web services to be extended with additional methods or overriding existing methods.

The recommended path for custom entry points is the following `custom/service/<version>_custom/`. At the time of writing the latest web service version is `v4_1` so this would be `custom/service/v4_1_custom/`.

Next we create the implementing class. This will create our new method. In our example we will simply create a new method which writes to the SuiteCRM log We will call this method `write_log_message`.

Example 16.5: Custom v4_1 Web Service Implementation

```php
<?php
if(!defined('sugarEntry')){
  define('sugarEntry', true);
}
require_once 'service/v4_1/SugarWebServiceImplv4_1.php';
class SugarWebServiceImplv4_1_custom extends SugarWebServiceImplv4_1
{

  function write_log_message($session, $message)
  {
    $GLOBALS['log']->info('Begin: write_log_message');

    //Here we check that $session represents a valid session
    if (!self::$helperObject->checkSessionAndModuleAccess(
                                                       $session,
                                                       'invalid_session',
                                                       '',
                                                       '',
                                                       '',
                                                       new SoapError()))
    {
      $GLOBALS['log']->info('End: write_log_message.');
      return false;
    }
    $GLOBALS['log']->info($message);
    return true;
  }
}
```

Next we create the registry file which will register our new method.

Example 16.6: Custom v4_1 web service registry

```php
<?php
    require_once 'service/v4_1/registry.php';
    class registry_v4_1_custom extends registry_v4_1
    {
        protected function registerFunction()
        {
            parent::registerFunction();
            $this->serviceClass->registerFunction('write_log_message',
                                                  array(
                                                    'session'=>'xsd:string',
                                                    'message'=>'xsd:string'),
                                                  array(
                                                    'return'=>'xsd:boolean')
                                                  );
        }
    }
```

Finally we create the entry point. This is the actual file that will be called by our API clients. This will reference the two files which we have created and will call the webservice implementation with our files.

Example 16.7: Custom v4_1 REST Entry point

```php
<?php
chdir('../../..');

require_once 'SugarWebServiceImplv4_1_custom.php';

$webservice_path = 'service/core/SugarRestService.php';
$webservice_class = 'SugarRestService';
$webservice_impl_class = 'SugarWebServiceImplv4_1_custom';
$registry_path = 'custom/service/v4_1_custom/registry.php';
$registry_class = 'registry_v4_1_custom';
$location = 'custom/service/v4_1_custom/rest.php';

require_once 'service/core/webservice.php';
```

Example 16.8: Custom v4_1 SOAP Entry point

```php
<?php
chdir('../../..');
require_once('SugarWebServiceImplv4_1_custom.php');
$webservice_class = 'SugarSoapService2';
$webservice_path = 'service/v2/SugarSoapService2.php';
$webservice_impl_class = 'SugarWebServiceImplv4_1_custom';
$registry_class = 'registry_v4_1_custom';
$registry_path = 'custom/service/v4_1_custom/registry.php';
$location = 'custom/service/v4_1_custom/soap.php';
require_once('service/core/webservice.php');
```

Usage

We can now use our custom endpoint. This is identical to using the API as detailed above, except that we use our custom entry point for either the SOAP WSDL or REST URL. For example using the same SuiteCRM location (`example.com/suitecrm`) as the above examples and using `v4_1`, we would use the following

Example 16.9: Custom v4_1 URLS

```
//SOAP WSDL
example.com/suitecrm/custom/service/v4_1_custom/soap.php?wsdl
//REST URL
example.com/suitecrm/custom/service/v4_1_custom/rest.php
```

17. Best Practices

Development instances

When making changes you should always use a development or test instance first. This allows you to fully and safely test any changes.

Version control

When developing customisations it is prudent to use some form of version control. Version control allows tracking changes to your codebase in addition to rolling back changes. There are many version control systems available. SuiteCRM uses Git[1] although I also like Mercurial[2].

If you are using a development instance (as mentioned above) then Version Control usually allows you to push changes to other versions or to tag releases. This provides a way of pushing changes to live or test instances safely. Crucially it also means that, should there be major problems with a version then this can be easily rolled back.

Backup

SuiteCRM has been developed to be customisable. However, mistakes, bugs and other unpleasantness can (and thanks to Murphy's law, will) happen. You should always ensure, before making any changes, that you have a backup of all files and of the database.

In order to back up files you can simply create a zip of the SuiteCRM directory and copy it to a safe place. On Linux systems this can be performed using the following:

Example 17.1: File backup

```
tar -czvf suitecrmfilebackup.tar.gz /path/to/suitecrm
```

Backing up the SuiteCRM database will vary depending on which database you are using. However MySQL backups can be performed using the `mysqldump` command on Linux as seen here:

[1]http://git-scm.com/

[2]http://mercurial.selenic.com/

Example 17.2: MySQL Database backup

```
mysqldump suitecrmdatabase -u databaseuser -p | gzip -c | cat > suitecrm.sql.gz
```

Be upgrade safe

Unless you are making changes to a custom module you should strive in all cases to use the custom framework and make changes in the custom folder. This ensures that, should you make a mistake, rectifying the mistake is as simple as removing the customisation.

However the main advantage to using `custom` is that, when you upgrade SuiteCRM in the future you will not have your changes overwritten by the updated SuiteCRM files. See the Extensions chapter for more information.

Use appropriate log levels

Using appropriate log levels (See the chapter on Logging) makes it easier to track down issues. You do not want very important messages to be logged as `debug` since this will make them difficult to find. Likewise you don't want unimportant log messages cluttering up the `fatal` log output.

Long running logic hooks

If a logic hook task is long running then you should place it into the job queue (see the Logic Hook and Scheduled Tasks chapters).

Minimise SQL

Where possible you should strive to use the SuiteCRM supplied methods of accessing data. This includes using beans and the BeanFactory where possible (see the chapter on Working with beans). There are a few reasons for this. The first is that SQL is usually either hardcoded or has to be dynamically built. In the case where the SQL is hardcoded this means that changes to fields will not be reflected thereby making your code more brittle.

Dynamic SQL is better because it can react to field changes and generally be tailored to fit the situation. However this requires adding extra, often complex, code. It can be hard to account for all situations (this can be especially problematic when attempting to traverse relationships dynamically).

Another issue is that, usually SQL will end up being Database specific (see the next point for mitigating this however).

Finally any custom logic (such as Logic Hooks) which would usually be fired for saving beans or relationships will not be fired for SQL queries.

SQL Use

In some cases using raw SQL is unavoidable. If that is the case then you should strive to use standard compliant SQL. If database engine specific features need to be used, and you wish to target other database engines, you can check for the DB type. For example:

Example 17.1: Checking for the database engine

```
function getSomeSQLQuery(){
    global $sugar_config;
    switch($sugar_config['dbconfig']['db_type']){
        case 'mssql':
            $sql = 'MSSQL specific SQL';
            break;
        case 'mysql':
        default:
            $sql = 'MySQL specific SQL';
            break;
    }
    return $sql;
}
```

Entry check

The majority of SuiteCRM files will start with some variation of the following line:

Example 17.2: Entry check

```
if(!defined('sugarEntry') || !sugarEntry) die('Not A Valid Entry Point');
```

This prevents direct access to the file by ensuring that SuiteCRM has been loaded through a valid entry point (i.e. it has been loaded through index.php, cron.php or a custom entry point).

Redirect after post

Sometimes you may have custom controller actions (see the controller section) or custom entry points (see the Entry Points chapter). These actions and entry points or other pages are usually accessed using POST. After a POST request it is a web best practice to redirect to a different page, especially if your page makes any changes. This prevents the user from refreshing the page and causing a duplicate action. Within SuiteCRM it is best to use the `SugarApplication::redirect` method to redirect. This simply accepts a URL. As follows:

Example 17.3: Redirecting within SuiteCRM

```
SugarApplication::redirect('index.php?module=<TheModule>');
```

18. Performance Tweaks

In most cases the performance of SuiteCRM should not be an issue. However in the case of large datasets or systems with many users you may notice some performance degradation. These changes can help improve performance.

Server

The server that SuiteCRM runs on is, of course, very important when it comes to the kind of performance you can expect. A full guide on server setup is outside the scope of this book. However there are some things you can do to ensure that you get the best performance out of SuiteCRM.

PHP

Installing a PHP opcode cache will increase the performance of all PHP files. These work by caching the compilation of PHP files resulting in less work on each request. Furthermore SuiteCRM will use the caching API of some PHP accelerators which will further increase performance. If you are using Linux then APC[1] is the usual choice. Windows users should check out WinCache[2].

MySQL

MySQL is notorious for having small default settings. Fully optimising MySQL is outside the scope of this book (however checkout mysqltuner.pl[3] for a helpful Perl script which will provide setting recommendations - note that you should be careful when running files from an unknown source). One small change that can make a big difference is increasing the `innodb_buffer_pool_size`.

If you have migrated or imported a significant amount of data it is possible that some tables will be fragmented. Running `OPTIMIZE TABLE tablename` can increase performance.

Indexes

Adding indexes on the fields of modules can improve database performance. The core modules usually have important fields indexed. However if you have created a new module or added new, often searched fields to a module then these fields may benefit from being indexed. See the Vardef chapter for adding indexes.

[1]http://php.net/manual/en/book.apc.php
[2]http://php.net/manual/en/book.wincache.php
[3]http://mysqltuner.pl

Config Changes

The following are some config settings that can be used to improve performance. Please note that in most cases you will have better performance gains by first following the steps in previous sections. These settings should be set in the config_override.php file. See the chapter on the Config files for more information.

```
$sugar_config['developerMode'] = false;
```

Unless you are actively developing on an instance developerMode should be off. Otherwise each page request will cause cached files to be reloaded.

```
$sugar_config['disable_count_query'] = true;
```

For systems with large amounts of data the count queries on subpanels used for the pagination controls can become slow thereby causing the page to be sluggish or outright slow to load. Disabling these queries can improve performance dramatically on some pages.

```
$sugar_config['disable_vcr'] = true;
```

By default opening the detail view of a record from the list view will also load the other records in the list to allow for easy moving through records. If you do not use this feature, or, if loading the detail view for some records has become slow, you can disable this feature.

```
$sugar_config['list_max_entries_per_page'] = '10';
```

The number of records shown in each page of the list view can be decreased. This will result in a slight increase in performance on list view pages.

```
$sugar_config['logger']['level'] = 'fatal';
```

Lowering the log level means that there will be less log messages to write to disk on each request. This will slightly (very slightly) increase performance.

19. Further Resources

Although this book has aimed to be a thorough resource, SuiteCRM is large and feature rich. Therefore it is not possible to include all the information you may require. Here are some extra resources for developing with SuiteCRM.

SuiteCRM Website

The SuiteCRM website (SuiteCRM.com[1] has many excellent resources including:

- SuiteCRM forums[2] - come and say hi!
- SuiteCRM Blog[3]
- SuiteCRM Wiki[4]

External SuiteCRM Resources

SuiteCRM GitHub[5]
The SuiteCRM source code is hosted on GitHub. Here you can get bleeding edge code changes and even contribute code.

SugarCRM Resources

SuiteCRM has strived to remain compatible with the SugarCRM community edition and much of the documentation is still valid. The appropriate version for SuiteCRM information is 6.5. Versions of documentation higher than this (i.e. 7) will probably not be relevant.

- SugarCRM Developer docs[6]
- SugarCRM Developer Blog[7]

Technical Links

- PHP[8] - The main language used by SuiteCRM

[1]http://suitecrm.com
[2]https://suitecrm.com/forum/index
[3]https://suitecrm.com/suitecrm/blog
[4]https://suitecrm.com/wiki/index.php/Main_Page
[5]https://github.com/salesagility/SuiteCRM
[6]http://support.sugarcrm.com/02_Documentation/04_Sugar_Developer/
[7]http://developer.sugarcrm.com/
[8]http://php.net/

- Smarty[9] - The templating language used throughout SuiteCRM.
- XDebug[10] - Debugging/profiling extension for PHP
- Git[11] - Distributed version control system
- YUI[12] - Legacy Javascript library used in SuiteCRM
- JQuery[13] - Javascript library used in SuiteCRM - to be preferred over YUI.
- PHPMailer[14] Email library used in SuiteCRM
- APC[15] - Alternative PHP Cache. PHP Opcode cache supported by SuiteCRM
- WinCache[16] - Windows PHP cache. PHP Opcode cache supported by SuiteCRM
- PHPStorm[17] - PHP IDE (Paid)
- Eclipse PHP Development Tools[18] - PHP IDE (Free and Open Source)

Other Links

- SalesAgility[19] - The company behind SuiteCRM.
- Jim Mackin[20] - Me :)

[9] http://www.smarty.net/
[10] http://xdebug.org
[11] http://git-scm.com/
[12] http://yuilibrary.com/
[13] https://jquery.com/
[14] https://github.com/PHPMailer/PHPMailer
[15] http://php.net/manual/en/book.apc.php
[16] http://php.net/manual/en/book.wincache.php
[17] https://www.jetbrains.com/phpstorm/
[18] https://eclipse.org/pdt/
[19] https://salesagility.com/
[20] http://www.jsmackin.co.uk

20. Appendix A - Code Examples

Metadata

This is an example of setting up a function subpanel (see the Metadata chapter for more information).

In this example the cases module has a custom field `incident_code_c` which is used to track cases with the same root cause. We'll add a subpanel to show all cases that have the same `incident_code_c`.

Initially we add to the `subpanel_setup` section of Cases by creating the following file in `custom/Extension/modules/Cases/Ext/Layoutdefs/IncidentLayoutdefs.php`

Example A.1: **`IncidentLayoutdefs.php`**

```php
<?php
$layout_defs["Cases"]["subpanel_setup"]['incident_cases'] = array(
  'module' => 'Cases',
  'title_key' => 'LBL_INCIDENT_CASES',
  'subpanel_name' => 'default',
  'get_subpanel_data' => 'function:get_cases_by_incident',
  'function_parameters' =>
          array('import_function_file' => 'custom/modules/Cases/IncidentUtils.ph\
p',),
);
```

Next we create the file which will define our `get_cases_by_incident` function `custom/modules/Cases/IncidentUtils.php`.

Example A.2: **`IncidentUtils.php`**

```php
<?php
function get_cases_by_incident(){
        global $db;
        //Get the current bean
        $bean = $GLOBALS['app']->controller->bean;
        $incidentCode = $db->quote($bean->incident_code_c);
        //Create the SQL array
        $ret = array();
        $ret['select'] = ' SELECT id FROM cases ';
        $ret['from'] = ' FROM cases ';
        $ret['join'] = "";
        //Get all cases where the incident code matches but exclude the current \
case.
        $ret['where']="WHERE cases.deleted = 0 AND cases_cstm.incident_code_c = \
'{$incidentCode}' AND cases.id != '{$bean->id}'";
        return $ret;
}
```

Module Installer

The following is a basic example manifest file. See the Module Installer chapter.

Example A.3: Example manifest file

```
$manifest = array(
  'name' => 'Example manifest',
  'description' => 'A basic manifest example',
  'version' => '1.2.3',
  'author' => 'Jim Mackin',
  'readme' => 'This is a manifest example for the SuiteCRM for Developers book',
  'acceptable_sugar_flavors' => array('CE'),
  'acceptable_sugar_versions' => array(
      'exact_matches' => array('6.5.20',),
  ),
  'dependencies' => array(
    array(
      'id_name' => 'hello_world',
      'version' => '3.2.1'
    ),
  ),
  'icon' => 'ManifestExample.png',
  'is_uninstallable' => true,
  'published_date' => '2015-05-05',
  'type' => 'module',
  'remove_tables' => 'prompt',
);
$installdefs = array(
  'id' => 'suitecrmfordevelopers_example_manifest',
  'image_dir' => '/images/',
  'copy' => array(
    array(
      'from' => '/modules/ExampleModule',
      'to' => 'modules/ExampleModule',
    ),
  ),
  'dashlets' => array(
    array(
      'from' => '/modules/ExampleModule/Dashlets/',
      'name' => 'ExampleModuleDashlet'
    )
  ),
  'language' => array(
    array(
      'from' => 'application/language/en_us.examplemoduleadmin.php',
      'to_module' => 'application',
      'language' => 'en_us'
    ),
    array(
      'from' => '/modules/Accounts/language/en_us.examplemodule.php',
      'to_module' => 'Accounts',
      'language' => 'en_us'
    ),
    array(
      'from' => '/application/language/es_es.examplemoduleadmin.php',
      'to_module' => 'application',
      'language' => 'es_es'
    ),
    array(
      'from' => '/modules/Accounts/language/es_es.examplemodule.php',
      'to_module' => 'Accounts',
      'language' => 'es_es'
    ),
  ),
  'custom_fields' => array(
    array(
      'name' => 'example_field',
      'label' => 'Example Field',
      'type' => 'varchar',
      'max_size' =>  100,
      'module' => 'Accounts',
    ),
  ),
  'vardefs' => array(
    array(
      'from' => 'modules/Accounts/vardefs/examplemodule_vardefs.php',
      'to_module' => 'Accounts',
    ),
  ),
  'beans' => array(
    array(
      'module' => 'ExampleModule',
      'class' => 'ExampleModule',
      'path' => 'modules/ExampleModule/ExampleModule.php',
    ),
  ),
  'logic_hooks' => array(
    array(
      'module' => 'Accounts',
      'hook' => 'before_save',
      'order' => 100,
      'description'  => 'Example module before save hook',
      'file' => 'modules/ExampleModule/ExampleModuleHook.php',
      'class' => 'ExampleModuleLogicHooks',
      'function' => 'accounts_before_save',
    ),
  ),
  'administration' => array(
    array(
      'from' => 'modules/administration/examplemodule_admin.php',
    ),
  ),
);
$upgrade_manifest = array(
);
```

21. Appendix B - API Methods

Methods

login

Logs into SuiteCRM and returns a session id used for subsequent API calls.

Arguments

login arguments

Name	Type	Desc
user_auth	array	Authentication details for the API User
user_auth[user_name]	string	The user name of the SuiteCRM user. Required.
user_auth[password]	string	The MD5 hash of the password for user_name. Required.
application_name	string	An identifier for the application accessing the API
name_value_list	name_value_list	An array of login options
name_value_list[language]	string	The language for this user
name_value_list[notifyonsave]	bool	Send email notifications when a new record is saved and assigned to a user

Response

login response

Name	Type	Desc
id	string	The session id for this login. Required for all other API calls.
name_value_list	name_value_list	An array containing information about this user.
name_value_list[user_id]	string	The id of the logged in user.
name_value_list[user_name]	string	The user_name of the logged in user
name_value_list[user_language]	string	The language setting of the logged in user
name_value_list[user_currency_id]	string	The id of the currency of the logged in user. -99 is the default currency.
name_value_list[user_currency_name]	string	The name of the currency of the logged in user.

login response

Name	Type	Desc
name_value_list[user_is_admin]	bool	Whether the logged in user is an admin
name_value_list[user_default_team_id]	string	The default team of the logged in user. This value comes from before the fork of SuiteCRM and isn't used.
name_value_list[user_default_-dateformat]	string	The default date format of the logged in user.
name_value_list[user_default_-timeformat]	string	The default time format of the logged in user
name_value_list[user_number_-seperator]	string	The number separator of the logged in user. (I.e. comma for numbers in the 1,000.00 format)
name_value_list[user_decimal_-seperator]	string	The decimal separator of the logged in user. (I.e. period for numbers in the 1,000.00 format)
name_value_list[mobile_max_list_-entries]	int	Max list entries for the logged in user (simply grabs the wl_list_max_entries_per_subpanel config key)
name_value_list[mobile_max_-subpanel_entries]	int	Max subpanel entries for the logged in user(simply grabs the wl_list_max_entries_per_subpanel config key)

logout

Logs the web user out of SuiteCRM and destroys the session.

Arguments

logout arguments

Name	Type	Desc
session	string	The session id. See login.

Response

No response.

get_available_modules

Returns a list of the modules available for use. Also returns the ACL (Access Control List) for each module.

Arguments

get_available_modules arguments

Name	Type	Desc
session	string	The session id. See login.
filter	string	Filter the modules returned. Either 'default', 'mobile' or 'all'.

Response

get_available_modules response

Name	Type	Desc
modules	array	An array containing the module details.
modules[][module_key]	string	The key for this module
modules[][module_label]	string	The label for this module
modules[][favorite_enabled]	bool	Favorites were SugarCRM Professional functionality. This is always empty.
modules[][acls]	array	An array containing the ACL list - that is what actions are allowed.
modules[][acls][][action]	string	The action i.e. edit, delete, list etc.
modules[][acls][][access]	bool	Whether or not access is allowed.

get_document_revision

Returns the details for a specific document revision.

Arguments

get_document_revision arguments

Name	Type	Desc
session	string	The session id. See login.
i	string	The id of the document revision to retrieve

Response

get_document_revision response

Name	Type	Desc
document_revision	array	An array containing the document revision details
document_revision[id]	string	The id of the document revision.
document_revision[document_name]	string	The name of the document revision
document_revision[revision]	int	The revision number of the document revision.
document_revision[filename]	string	The filename of the file
document_revision[file]	binary string	The full contents of the file

get_entries

Gets a list of entries for a specific module and list of module ids. Optionally allows returning related records.

Arguments

get_entries arguments

Name	Type	Desc
session	string	The session id. See login.
module_name	string	The name of the module to display entries for.
ids	array	An array of record ids to fetch
ids[]	string	An individual id
select_fields	array	An array of fields to return. An empty array will return all fields.
select_fields[]	string	The name of a field to return
link_name_to_fields_array	name_value_list	An array of relationships to retrieved.
link_name_to_fields_array[][name]	string	The name of the link to follow (as defined in `module_name`).
link_name_to_fields_array[][value]	array	An array of the fields to return for this related module.
link_name_to_fields_array[][value][]	string	The field name
track_view	bool	Whether to mark these records as recently viewed.

Response

get_entries response

Name	Type	Desc
entry_list	array	An array of records.
entry_list[]	array	Details for an individual record.
entry_list[][id]	string	The id of this record.
entry_list[][module_name]	string	The name of the module this record belongs to.
entry_list[][name_value_list]	name_value_list	An array containing each returned field.
entry_list[][name_value_list][]	array	Details for an individual field.
entry_list[][name_value_list][][name]	string	The name of the field.
entry_list[][name_value_list][][value]	string	The value of the field.
relationship_list	array	An array of arrays containing the relationships for the corresponding record.
relationship_list[]	array	The relationships for the corresponding record.
relationship_list[link_list]	array	The list of relationships for this record.
relationship_list[link_list][]	array	Details of a single relationship.
relationship_list[link_list][][name]	string	The name of this relationship.
relationship_list[link_list][][records]	array	The related records for this relationship.
relationship_list[link_list][][records][]	array	Details of a single related record.
relationship_list[link_-list][][records][][link_value]	name_value_list	An array of the requested fields for this relationship.
relationship_list[link_-list][][records][][link_value][]	array	A name value pair for this particular field.
relationship_list[link_-list][][records][][link_value][name]	string	The name of the field.
relationship_list[link_-list][][records][][link_value][value]	string	The value of the field.

get_entries_count

Returns a count of entries matching the given query.

Arguments

get_entries_count arguments

Name	Type	Desc
session	string	The session id. See login.
module_name	string	The name of the module to display entries for.
query	string	An SQL WHERE clause to apply to the query.
deleted	bool	Whether to include deleted records

Response

get_entries_count response

Name	Type	Desc
result_count	int	The count of matching entries.

get_entry

Returns the details for a single record. Optionally allows returning related records.

Arguments

get_entry arguments

Name	Type	Desc
session	string	The session id. See login.
module_name	string	The name of the module to fetch the entry for.
id	string	The id of the record to fetch
select_fields	array	An array of fields to return. An empty array will return all fields.
select_fields[]	string	The name of a field to return
link_name_to_fields_array	name_value_list	An array of relationships to retrieved.
link_name_to_fields_array[][name]	string	The name of the link to follow (as defined in `module_name`).
link_name_to_fields_array[][value]	array	An array of the fields to return for this related module.
link_name_to_fields_array[][value][]	string	The field name
track_view	bool	Whether to mark these records as recently viewed.

Response

Identical to the response by `get_entries` except only one record will be returned.

`get_entry_list`

Arguments

get_entry_list arguments

Name	Type	Desc
session	string	The session id. See login.
module_name	string	The name of the module to fetch the entry for.
query	string	An SQL WHERE clause to apply to the query.
order_by	string	In theory for ordering results but this is unused.
offset	int	The result offset. Useful for pagination.
select_fields	array	An array of fields to return. An empty array will return all fields.
select_fields[]	string	The name of a field to return
link_name_to_fields_array	name_value_list	An array of relationships to retrieved.
link_name_to_fields_array[][name]	string	The name of the link to follow (as defined in `module_name`).
link_name_to_fields_array[][value]	array	An array of the fields to return for this related module.
link_name_to_fields_array[][value][]	string	The field name
max_results	int	The maximum number of results to return. Useful for pagination.
deleted	bool	Whether to include deleted records.
favorites	bool	Favorites were SugarCRM Professional functionality. This is unused.

Response

get_entry_list response

Name	Type	Desc
result_count	int	The number of returned records.
total_count	int	The total number of records matching the query.
next_offset	int	The offset of the next set of records.
entry_list	array	An array of records.
entry_list[]	array	Details for an individual record.
entry_list[][id]	string	The id of this record.
entry_list[][module_name]	string	The name of the module this record belongs to.
entry_list[][name_value_list]	name_value_list	An array containing each returned field.
entry_list[][name_value_list][]	array	Details for an individual field.
entry_list[][name_value_list][][name]	string	The name of the field.
entry_list[][name_value_list][][value]	string	The value of the field.
relationship_list	array	An array of arrays containing the relationships for the corresponding record.

get_entry_list response

Name	Type	Desc
relationship_list[]	array	The relationships for the corresponding record.
relationship_list[link_list]	array	The list of relationships for this record.
relationship_list[link_list][]	array	Details of a single relationship.
relationship_list[link_list][][name]	string	The name of this relationship.
relationship_list[link_list][][records]	array	The related records for this relationship.
relationship_list[link_list][][records][]	array	Details of a single related record.
relationship_list[link_-list][][records][][link_value]	name_value_list	An array of the requested fields for this relationship.
relationship_list[link_-list][][records][][link_value][]	array	A name value pair for this particular field.
relationship_list[link_-list][][records][][link_value][name]	string	The name of the field.
relationship_list[link_-list][][records][][link_value][value]	string	The value of the field.

get_language_definition

Returns

Arguments

get_language_definition arguments

Name	Type	Desc
session	string	The session id. See login.
modules	array	An array of the modules to return language labels for
modules[]	string	The modules name.
md5	bool	Whether to return the md5 for each module. Can be useful for caching responses.

Response

get_language_definition response

Name	Type	Desc
result[	string/array	An array of the labels or an md5 string for

`get_last_viewed`

Returns a list of the most recently viewed modules for the current user.

Arguments

get_last_viewed arguments

Name	Type	Desc
session	string	The session id. See login.
module_names	array	An array of the modules to return the last viewed records for.
module_names[]	string	The modules name.

Response

get_last_viewed response

Name	Type	Desc
result[]	array	An array of the details of recently viewed records
result[][id]	int	The id of the tracker row for this viewing
result[][item_id]	string	The id of the viewed record.
result[][item_summary]	string	The summary of the record. This is usually it’s name.
result[][module_name]	string	The module for this record.
result[][monitor_id]	string	The monitor id for this viewing. Legacy and unused.
result[][date_modified]	string	The date that this record was viewed.

`get_modified_relationships`

Returns a list of the modified relationships for the current user between one of the Calls, Meetings or Contacts modules.

Arguments

get_modified_relationships arguments

Name	Type	Desc
session	string	The session id. See login.
module_name	string	The name of the module to retrieve relationships for. Always `Users`.
related_module	string	The related module to retrieve records for. One of `Meetings`, `Calls` or `Contacts`.
from_date	string	The start date of the range to search. In the format `Y-m-d H:i:s`.
to_date	string	The end date of the range to search. In the format `Y-m-d H:i:s`.
offset	int	The record offset to start with.
max_results	int	The maximum number of results to return.
deleted	bool	Whether to include deleted records.
module_user_id	string	In theory the id of the user to return relationships for. However the current user is always used.
select_fields	array	An array of the fields to return for the relationship record. An empty array will return all fields.
select_fields[]	string	The name of the field to return.
relationship_name	string	The name of the relationship between `module_name` and `related_module`.
deletion_date	string	A start date for the range in which to return deleted records. In the format `Y-m-d H:i:s`.

Response

get_modified_relationships response

Name	Type	Desc
result_count	int	The number of returned records.
next_offset	int	The offset of the next set of records.
entry_list	array	An array of the returned records.
entry_list[]	array	Details for an individual record.
entry_list[][id]	string	The id of this record.
entry_list[][module_name]	string	The name of the module this record belongs to.
entry_list[][name_value_list]	name_value_list	An array containing each returned field.
entry_list[][name_value_list][]	array	A name value pair of the field information.
entry_list[][name_value_list][][name]	string	The name of the field.
entry_list[][name_value_list][][value]	string	The value of the field.
error	array	An array containing the error details.
error[number]	int	The error number of the error that occurred.
error[name]	string	The name of the error that occurred.
error[description]	string	A description of the error that occurred.

`get_module_fields`

Returns the field definitions for a given module.

Arguments

get_module_fields arguments

Name	Type	Desc
session	string	The session id. See login.
module_name	string	The name of the module to return field definitions for.
fields	array	An array of fields to return definitions for. An empty array will return all fields.
fields[]	string	The name of the field.

Response

get_module_fields response

Name	Type	Desc
module_name	string	The name of the module.
table_name	string	The name of the database table for this module.
module_fields	array	An array of the requested fields for this module.
module_fields[]	array	The details of a specific field.
module_fields[name]	string	The name of the field.
module_fields[type]	string	The type of the field.
module_fields[group]	string	The group of fields that this field belongs to. Used for addresses or link definitions.
module_fields[id_name]	string	The name of the id field on this module for this link if appropriate.
module_fields[label]	string	The display label for this field.
module_fields[required]	bool	Whether this field is required or not.
module_fields[options]	name_value_list	An array of possible options for this field. An empty array if options are not appropriate for this field type.
module_fields[options][]	array	A name value pair of a single option.
module_fields[options][][name]	string	The options key.
module_fields[options][][value]	string	The options display value.
module_fields[related_module]	string	The related module for this field if it is a related type. Empty otherwise.
module_fields[calculated]	string	Calculated fields were a SugarCRM professional feature. Will be empty.
module_fields[len]	int	The length of this field or an empty string if this is not appropriate for this field type.
link_fields	array	An array of the requested link fields for this module.
link_fields[]	array	The details of a specific field.
link_fields[name]	string	The name of the field.
link_fields[type]	string	The type of the field. Will always be link.

get_module_fields response

Name	Type	Desc
link_fields[group]	string	The group of fields that this field belongs to. Will be empty for links.
link_fields[id_name]	string	The name of the id field on this module for this link if appropriate.
link_fields[relationship]	string	The relationship name for this link.
link_fields[module]	string	The module this field links to.
link_fields[bean_name]	string	The bean that this field links to.

`get_module_fields_md5`

Returns an md5 of the a modules field definitions. Useful for caching.

Arguments

get_module_fields_md5 arguments

Name	Type	Desc
session	string	The session id. See login.
module_names	array	An array of modules to return the md5 for.
module_names[]	string	The name of the module to return the field definitions md5 for.

Response

get_module_fields_md5 response

Name	Type	Desc
result[]	array	An array of the md5's keyed by the module name.
result[	string	The md5 string for

`get_module_layout`

Returns the layout for specified modules and views. Optionally returns an md5 of the layouts.

Arguments

get_module_layout arguments

Name	Type	Desc
session	string	The session id. See login.
modules	array	An array of the modules to return layouts for.
modules[]	string	The name of the module.
types	array	An array of the types of views to return. Only `default` is supported.
types[]	string	The type of the views.
views	array	An array of the views to return. One of `edit`, `detail`, `list` and `subpanel`.
views[]	string	The name of the view.
acl_check	bool	Whether or not to check that the current user has access to this module and view.
md5	bool	Whether or not to return the view as an md5 string. Useful for caching.

Response

get_module_layout response

Name	Type	Desc
result	array	The array of results keyed by module name.
result[	array	An array of layouts for
result[	array	An array of the layouts keyed by the view name.
result[	array/string	The layout of the view

`get_module_layout_md5`

Returns the md5 of the specified views for the specified modules. Behaves identically to get_module_layout with the md5 parameter set to true.

Arguments

get_module_layout arguments

Name	Type	Desc
session	string	The session id. See login.
modules	array	An array of the modules to return layouts for.
modules[]	string	The name of the module.
types	array	An array of the types of views to return. Only `default` is supported.
types[]	string	The type of the views.
views	array	An array of the views to return. One of `edit`, `detail`, `list` and `subpanel`.
views[]	string	The name of the view.
acl_check	bool	Whether or not to check that the current user has access to this module and view.

Response

get_module_layout_md5 response

Name	Type	Desc
md5	array	The array of results keyed by module name.
md5[	array	An array of layouts for
md5[	array	An array of the layouts keyed by the view name.
md5[	string	The md5 of the layout layout of the view

get_relationships

Returns related records given a specific module, record and list of links. ####Arguments

get_relationships arguments

Name	Type	Desc
session	string	The session id. See login.
module_name	string	The module to return relationships for.
module_id	string	The record to return relationships for.
link_field_name	string	The link field to follow for this record.
related_module_query	string	A WHERE clause to use to filter the related modules by.
related_fields	array	An array of the fields to return for matching records.
related_fields[]	string	The name of the field.
related_module_link_name_to_fields_-array	name_value_list	An array of related fields to return for matching records.
related_module_link_name_to_fields_-array[]	array	Details for a specific link.
related_module_link_name_to_fields_-array[][name]	string	The name of the link to follow for matching records.
related_module_link_name_to_fields_-array[][value]	array	An array of fields to return for this link.
related_module_link_name_to_fields_-array[][value][]	string	The field name.
deleted	bool	Whether to include deleted records.
order_by	string	In theory for ordering results but this is unused.
offset	int	The record offset to start with.
limit	int	The maximum number of results to return.

Response

Identical to the response by `get_entries`.

`get_server_info`

Returns information about the SuiteCRM server. Currently still returns information about the SugarCRM flavor and versions.

Arguments

No arguments.

Response

get_server_info response

Name	Type	Desc
flavor	string	The SugarCRM flavor. For SuiteCRM will always be 'CE'.
version	string	The SugarCRM version. Note this this is distinct from the SuiteCRM version
gmt_time	string	The server time in UTC.

`get_upcoming_activities`

Returns a list of the 10 upcoming activities (Meetings, Calls and Tasks - also includes Opportunities) for the currently logged in user.

Arguments

get_upcoming_activities arguments

Name	Type	Desc
session	string	The session id. See login.

Response

get_upcoming_activities response

Name	Type	Desc
result	array	An array of the upcoming activities.
result[]	array	The details of a single activity.
result[][id]	string	The id of this activity.
result[][module]	string	The module for this activity.
result[][date_due]	string	The due date for this activity.
result[][summary]	string	The summary of this activity. Usually simply it's name.

`get_user_id`

Returns the id of the currently logged in user.

Arguments

get_user_id arguments

Name	Type	Desc
session	string	The session id. See login.

Response

get_user_id response

Name	Type	Desc
id	string	The id of the current user.

`seamless_login`

Marks a session as allowing a seamless login. If successful then the session id (see the login call) can be used in a URL (as MSID) to log the user into SuiteCRM in the browser seamlessly. For example if you have the session id `1234` then accessing the URL `example.com/index.php?MSID=1234`. The MSID parameter can be used in any valid SuiteCRM URL.

Arguments

seamless_login arguments

Name	Type	Desc
session	string	The session id. See login.

Response

seamless_login response

Name	Type	Desc
result	bool	Boolean indicating success

`search_by_module`

Allows searching for records that contain a specific search string.

Arguments

search_by_module arguments

Name	Type	Desc
session	string	The session id. See login.
search_string	string	The string to search for.
modules	array	An array of the modules to include in the search.
modules[]	string	The modules name.
offset	int	The result offset. Useful for pagination.
max_results	int	The maximum number of results to return. Useful for pagination.
assigned_user_id	string	Filter by the given assigned user. Leave blank to do no user filtering.
select_fields	array	An array of the fields to return for the found records. An empty array will return all fields.
select_fields[]	string	The name of the field to return.
unified_search_only	bool	Whether to only return records for modules that participate in the global search.
favorites	bool	Favorites were SugarCRM Professional functionality. This is unused.

Response

search_by_module response

Name	Type	Desc
entry_list	array	An array of the results for each module.
entry_list[]	array	Results for a specific module.
entry_list[][name]	string	The name of the module that this entry contains results for.
entry_list[][records]	array	An array of the record results.
entry_list[][records][]	name_value_list	A name value list of records id and name.
entry_list[][records][][id]	array	A name value pair containing the id of this record.
entry_list[][records][][name]	array	A name value pair containing the name of this record.

set_document_revision

Creates a new document revision for a document.

Arguments

set_document_revision arguments

Name	Type	Desc
session	string	The session id. See login.
note	array	An array containing the document revision details.
note[id]	string	The id of the document to add this revision to.
note[file]	binary string	The binary contents of the file, base 64 encoded.
note[filename]	string	The name of the file.
note[revision]	int	The revision number for this revision.

Response

set_document_revision response

Name	Type	Desc
id	string	The id of the newly created document revision.

set_entries

Creates or updates a list of records.

Arguments

Note: Supplying a value for the id field will perform an update for that record.

set_entries arguments

Name	Type	Desc
session	string	The session id. See login.
module_name	string	The name of the module to create/update records for.
name_value_lists	name_value_list	An array of the details for each record to create/update.
name_value_lists[]	array	Details of an individual record.
name_value_lists[][]	array	A name value pair for each field value.
name_value_lists[][][name]	array	The name of the field.
name_value_lists[][][value]	array	The value for the field.

Response

set_entries response

Name	Type	Desc
ids	array	An array of the resulting ids. Returned in the same order as specified in the call to `set_entries`.
ids[]	array	The id for this record.

set_entry

Creates or updates a single record.

Arguments

Note: Supplying a value for the id field will perform an update for that record.

set_entries arguments

Name	Type	Desc
session	string	The session id. See login.
module_name	string	The name of the module to create/update a record for.
name_value_list	name_value_list	An array of the fields for the new/updated record.
name_value_lists[]	array	A name value pair for each field value.
name_value_lists[][name]	array	The name of the field.
name_value_lists[][value]	array	The value for the field.

Response

set_entries response

Name	Type	Desc
id	string	The id of the newly created or updated record.

get_note_attachment

Returns the details of a given note attachment.

Arguments

get_note_attachment arguments

Name	Type	Desc
session	string	The session id. See login.
id	string	The id of the note to retrieve information for.

Response

get_note_attachment response

Name	Type	Desc
note_attachment	array	The details for the note attachment.
note_attachment[id]	string	The id of the note to retrieve information for.
note_attachment[filename]	string	The filename of the file
note_attachment[file]	binary string	The full contents of the file
note_attachment[related_module_id]	string	The id of the record that this attachment is related to.
note_attachment[related_module_name]	string	The module of the record that this attachment is related to.

set_note_attachment

Creates a not attachment for a specified record.

Arguments

set_note_attachment arguments

Name	Type	Desc
session	string	The session id. See login.
note	array	The details for the note attachment.
note[id]	string	The id of the note to add an attachment for.
note[filename]	string	The filename of the file
note[file]	binary string	The full contents of the file base 64 encoded.

Response

set_entries response

Name	Type	Desc
id	string	The id of the note for this attachment.

set_relationship

Sets a relationship between a record and other records.

Arguments

set_relationship arguments

Name	Type	Desc
session	string	The session id. See login.
module_name	string	The name of the module to relate records to.
module_id	string	The id of the record to relate records to.
link_field_name	string	The name of the link field on the module through which records will be related.
related_ids	array	An array of record ids to relate.
related_ids[]	string	The id of a record to relate.
name_value_list	name_value_list	A name value list of additional relationship fields to set.
name_value_list[]	array	A name value pair for a relationship field to set.
name_value_list[][name]	string	The name of the field to set.
name_value_list[][value]	string	The value of the field to set.
delete	bool	Whether or not to delete the specified relationship instead of creating/updating it.

Response

set_relationship response

Name	Type	Desc
created	int	The number of relationships created.
failed	int	The number of relationships that failed to be created/deleted.
deleted	int	The number of relationships deleted.

set_relationships

Sets relationships between multiple records.

Arguments

set_relationships arguments

Name	Type	Desc
session	string	The session id. See login.
module_names	array	An array of modules to relate records to.
module_names[]	string	The name of the module to relate records to.
module_ids	array	An array of the ids of records to relate records to.
module_ids[]	string	The id of the record to relate records to.
link_field_names	string	An array of the link names through which records will be related.
link_field_names[]	string	The name of the link field on the module through which records will be related.
related_ids	array	An array of an array of record ids for each module specified.
related_ids[]	array	An array of record ids for the corresponding module.
related_ids[][]	string	The record id.
name_value_lists	array	An array of an array of name value list of additional relationship fields to set.
name_value_lists[]	name_value_list	An array of name value pairs for the relationship fields of the corresponding module.
name_value_lists[][name]	string	The name of the field to set.
name_value_lists[][value]	string	The value of the field to set.
delete_array	array	An array of booleans indicating whether or not the relationship should be deleted for each module.
delete_array[]	bool	Whether or not to delete the specified relationship instead of creating/updating it.

Response

set_relationships response

Name	Type	Desc
created	int	The number of relationships created.
failed	int	The number of relationships that failed to be created/deleted.
deleted	int	The number of relationships deleted.