

## 9.0 Developing With info@hand

When implementing a CRM, some of the key steps include:

1. Installing the CRM
2. Migrating and cleaning CRM data
3. Using the Studio to make small or medium adjustments, such as hiding un-needed modules, renaming fields, adding custom fields, re-arranging edit, detail and list view screen layouts, etc..
4. Integrating the CRM with other business systems, using the web services API offered by the CRM
5. Performing major customizations, or add entirely new modules, to accommodate business-specific requirements
6. User Training
7. Go Live!

So far in this guide we have not dealt with items 4 and 5 above, and that is the purpose of this section.

### 9.1 Integration Using Web Services

A number of our clients have been interested to use a variety of SugarCRM add-on products from third-party vendors, since the **info@hand** core CRM was originally built (starting in 2004) on a base of SugarCRM Open Source.

One of the key issues is the use of third party software that was designed to link with SugarCRM using SOAP or REST web services interfaces. The current revision of **info@hand** includes very little residual software from the SugarCRM Open Source project. However, it has been engineered to be closely compatible to the SOAP and REST APIs of SugarCRM CE release 6.2.

When a third party software uses a SOAP or REST call to **info@hand** to ask for the version of SugarCRM software, **info@hand** replies with this version info (6.2) by default. If you wish for some reason to change this answer, you may do so, by over-riding it in the *config.php* file. This is done by adding a configuration variable to the *config.php* file in this form:

```
'override_soap_version' => '4.5.1g',
```

This would set the reported SugarCRM version to 4.5.1g. This ability to over-ride the reported SugarCRM version can be useful to maintain compatibility with software such as Outlook and ThunderBird plugins that support SugarCRM Community Edition via a SOAP connection.

If a third-party module integrates with SugarCRM 6.2 solely by means of the SOAP or REST API, then there is a very good chance it will also work just fine with **info@hand**.

If you want to write your own software which accesses **info@hand** via the SOAP or REST APIs, you should follow the SugarCRM documentation found [here](#).

## 9.2 SugarCRM Module Compatibility

Third party software designed to install on SugarCRM Community or Professional/Enterprise Editions via the Upgrade Wizard or the Module Loader will need very significant editing to function with **info@hand**. The XML manifest file specifies the version and flavour of SugarCRM with which it is compatible, and this would need to be changed, at minimum, to load the software onto **info@hand**. However, the module will need furthering editing by a competent PHP software developer in order to be fully compatible with **info@hand**. The module architecture of **info@hand** 7.0 or later releases is entirely different from that of SugarCRM. The Long Reach Corporation, via one of our **info@hand** Partner organizations, can assist you with this sort of development work if required.

Be sure never to load a SugarCRM module into a live implementation of **info@hand** before it has been tested by a competent professional.

## 9.3 info@hand Module Development

### ➔ 9.3.1 Introduction

The process for developing custom modules for **info@hand** 7 is very different compared to previous versions, due to a set of fundamental changes in the **info@hand** base framework. Modules are more self-contained and their code contains fewer redundancies. A new configuration format is used in place of various PHP code files, meaning fewer opportunities for uncontrolled fatal errors. HTML templates are no longer used to define module `DetailView` or `EditView` forms, with new layout descriptors taking their place. Finally, separate model and display descriptor files replace `vardefs.php` for defining the structure of real and virtual database columns.

### ➔ 9.3.2 Configuration Files

The new configuration file format (`IAHConfig`) is a simple hierarchical format similar to YAML. Defining arrays of data, primarily character strings, it is easily parsed and written programmatically and designed to be human-editable as well.

#### *The IAHConfig Format*

By default, each line of the file specifies a key in an array. When the key is not followed by a colon, the value associated with it is assumed to be another array. Hard tabs are normally used to indicate depth although a sequence of four spaces is considered equivalent. The first line of the file generally consists of a PHP snippet which is not interpreted by the configuration system, but serves to protect the file contents.

```
<?php return; /* no output */ ?>
Key1
    Key2
# => array("Key1" => array("Key2" => array()))
```

When the key is followed by a colon, the value is represented by either a quoted string or an unquoted value, which may evaluate to a string or a special value. The following value may also be written over multiple (indented) lines, in which case the result is obtained by removing the indentation and trimming the string. Within quoted strings, the backslash character may be used for C-style character escaping. Special values include integer, float, and boolean literals (`true` and `false`) as well as `null`. Finally, array literals can be written as special values by using the square bracket format below.

```
String: test
Integer: 42
Float: 3.1416
Boolean: true
Array: [1, unquoted, "test\t\n"]
Multiline:
    A string
    on multiple lines.
# => array("String" => "test", "Integer" => 42, "Float" => 3.1416,
#       "Boolean" => true, "Array" => array(1, "unquoted", "test\t\n"),
#       "Multiline" => "A string\non multiple lines")
```

Because unquoted values are automatically trimmed, strings having leading or trailing whitespace need to be quoted. The special key value `-` (hyphen) represents the next numeric key, equivalent to setting `$result[]`, and is followed by a simple value or array literal. The special key value `--` (double hyphen) begins a new nested array at the next numeric key, and should be written alone on a line:

```
- Value1
--
    Key: Value2
# => array("Value1", array("Key" => "Value2"))
```

IAHConfig files may be easily parsed and written using the `ConfigParser` and `ConfigWriter` classes located in `/include/config/format`.

### ➔ 9.3.3 Module Directory Structure

Each module directory (subdirectory of `/modules`) follows a common directory structure. In the root of the directory, there is expected to be at least one PHP file containing a class deriving `SugarBean`. This is the primary bean class. Modules may contain more than one `SugarBean` class, with additional classes being more limited in their functionality (they won't be displayed in the Recently Viewed menu, cannot be referenced by `ref` fields, and have other restrictions). Each of these classes is also mapped to a `Model`, which describes the database mapping for this class. See `Model Descriptors` for details on these files, in particular the `bean_file` attribute on bean models.

Unlike in previous **info@hand** versions, these `SugarBean` classes are not used in most operations. Although the `retrieve()/save()` pattern may still be used, the preferred pattern is to perform

insertions and updates using a `RowUpdate` object for the target record. This method requires less memory and eliminates the formatting and un-formatting of field values for display.

The module directory will also contain several subdirectories:

<code>display</code>	This directory contains any Model Display Descriptor files. See section 9.3.6 for more information.
<code>language</code>	The location for any supporting language files for this module. See section 9.3.5 for more information.
<code>metadata</code>	Currently, the only file required in this directory is <code>module_info.php</code> , described below.
<code>models</code>	This directory contains any Model Descriptor files. See section 9.3.4 for more information.
<code>views</code>	This is the location for any Layout Descriptor files. See section 9.3.7 for more information.

The majority of these are explained in separate sections. For now, let's examine the `module_info.php` file located in the `metadata/` subdirectory. This file is required in order to let the system discover the primary SugarBean class and to display a tab for the module.

#### *Sample module\_info.php contents for the Contacts module*

```
detail
  primary_bean: Contact
  tab_visibility: normal
  default_group: LBL_TABGROUP_SALES_MARKETING
```

Inside the `detail` array there are 3 required attributes. `primary_bean` is the name of the primary model, a `bean` model descriptor, which will then provide the path to the primary class file. The `tab_visibility` attribute defines the display mode for the module tab: `normal`, indicating that a module tab should always be shown; `hidden`, meaning it should never be shown; and `manual`, if the tab should be shown only when specifically added to the system tab layout. In most cases this value should be `normal`, while supporting modules may use `hidden` to avoid cluttering the menu system. The last attribute, `default_group`, defines the tab group this module tab should be placed under. The tab may still be placed into another group by an administrator editing the system tab layout. If no default is provided and the tab visibility is `normal`, then it will be placed in whichever tab group contains the Administration module.

## ➔ 9.3.4 Model Descriptors

Previously represented by `vardefs.php`, in **info@hand 7** model descriptors are split into multiple files located in the `models/` subdirectory of the module directory, with additional system-level model

descriptors are located in `/include/models`. These are further classified as `bean`, `link`, and `table` descriptors with each generally representing a single database table. These files are automatically indexed by the `ModelManager` class with cached results written to `/cache/system/model_cache.php`, and their names are expected to be unique within the system. The Database Repair task is used to update the database definition according to these descriptors, creating tables, columns and indexes as required.

Of the standard model descriptor types, `bean` descriptors represent the common case. These are linked to a `SugarBean`-derived class, can be referenced by other `bean` and `link` descriptors (using `ref` fields), and use an `id` field as the primary index. Next are the `link` descriptors, which define tables representing many-to-many relationships between `bean` records. These tables may contain additional fields, known as relationship role columns. In order to prevent duplicate records in these tables, the primary key is usually composed of the two `id` columns defining the relationship. Finally, `table` descriptors map to general-purpose SQL tables with no default behaviour. Operations on these tables must be defined explicitly and auditing is not supported.

Each file must define a `detail` section, with properties that vary according to the descriptor type. These include:

<code>type</code>	The descriptor type, also used in the filename prefix.
<code>bean_file</code>	In <code>bean</code> descriptors, the path to the <code>SugarBean</code> -derived class file represented by this model.
<code>primary_key</code>	The column or columns used to create the table's primary key.
<code>table_name</code>	The (unique) name of the table as represented in the SQL database.
<code>comment</code>	A text comment describing the function of the table.
<code>default_order_by</code>	The default column used in sorting <code>ListView</code> results.
<code>display_name</code>	The column used to represent the displayed name of this record, for example when pointed to by a <code>ref</code> field or shown on the recently-viewed menu.

Various optional flags are available to configure system features for `bean`-type models:

<code>audit_enabled</code>	Enables auditing of database updates. Updates to fields also marked <code>audited</code> will be written to a separate audit table, along with the previous value, the time of the change, and the user ID performing the update.
<code>duplicate_merge</code>	To enable duplicate merging when a new record request appears similar to an existing record.
<code>optimistic_locking</code>	To enable optimistic locking for updates to this module.

importable	To allow mass importing of records into this module via the ImportDB interface. The value may be a string representing the name of a custom label (language string) for the import action.
reportable	Whether to allow Reports to be created and run against this model.
unified_search	To display this model (if it is the module primary bean) in the system Unified Search. Fields also marked <code>unified_search</code> will be used to automatically filter relevant results.

### Business Logic Hooks

Model descriptors may also define a `hooks` array containing a mapping of function hook definitions to be invoked when certain actions are performed. A function hook definition is itself an array, with most hooks defining only a `class_function` attribute. This is the name of a static class method on the SugarBean class referred to by this model (the attribute `class` may be set in order to override the containing class name). In place of `class_function`, the attribute `function` may be used to refer to a non-class function. In this case the attribute `file` should contain the path of the file containing this method (to be included once as needed).

#### *Sample business logic hooks defined by the Cases model*

```
hooks
  new_record
    --
    class_function: init_record
  notify
    --
    class_function: send_notification
    required_fields: [cust_contact_id]
```

Logic hooks may also define a `required_fields` attribute containing an array of field names. Fields added to this list will be automatically queried before the hook is executed so that their current values are available to the function.

Several logic hooks are currently supported:

<pre>new_record (     RowUpdate &amp;\$update,     array \$input )</pre>	<p>This hook is called in order to populate a new row, both before displaying the EditView form and after that form has been submitted. It is also executed for records created from external APIs (SOAP/JSON). The function may examine request parameters and update fields accordingly; it is most often used when creating a new record based on a related record in another module (in which case the related ID will be passed as a request parameter).</p>
<pre>before_save (     RowUpdate &amp;\$update ) after_save (     RowUpdate &amp;\$update )</pre>	<p>These hooks are executed for every row update. They may be used as a last chance to enforce class invariants and check user input, and to manage updates to related resources. The field updates to be performed may be accessed via the <code>\$updates</code> property of the <code>RowUpdate</code> object.</p> <p>A <code>before_save</code> hook may throw an <code>IAHActionCompleted</code> exception to indicate that the record update has been completed and the default behaviour must be skipped. An <code>IAHActionAbort</code> exception indicates that certain conditions have not been met and the record update cannot be completed.</p>
<pre>notify</pre>	<p>This hook is called after a successful <code>save</code> operation in order to send notification emails or otherwise alert users to the changes.</p>
<pre>before_delete (     RowUpdate &amp;\$update ) after_delete (     RowUpdate &amp;\$update )</pre>	<p>These hooks are called when a record is to be deleted (by setting <code>deleted=1</code> in the record, not removing it from the table).</p> <p>Like the <code>before_save</code> hook, the <code>before_delete</code> hook may throw <code>IAHActionCompleted</code> OR <code>IAHActionAbort</code>.</p>
<pre>before_add_link (     RowUpdate &amp;\$update ) after_add_link (     RowUpdate &amp;\$update )</pre>	<p>These hooks are executed when a record is being added or updated in a <code>link</code> model table. The details of the relationship data may be accessed via the <code>\$link_update</code> property of the <code>RowUpdate</code> object. This hook is called for the models on both sides of the relationship.</p> <p>Like the <code>before_save</code> hook, the <code>before_add_link</code> hook may throw <code>IAHActionCompleted</code> OR <code>IAHActionAbort</code>.</p>
<pre>before_remove_link (     RowUpdate &amp;\$update ) after_remove_link (     RowUpdate &amp;\$update )</pre>	<p>Called when a relationship between two records is being removed. Like the <code>before_save</code> hook, the <code>before_remove_link</code> hook may throw <code>IAHActionCompleted</code> OR <code>IAHActionAbort</code>.</p>

### Field Descriptors

The `fields` section of a model descriptor file contains a set of arrays describing the database columns. This is much like the `fields` section of earlier `vardefs.php` files. Each array key must be unique and represents either the name of the column or a reference to a system-defined field descriptor (shown below). Properties defined inside the array control the behaviour of the field. A non-exhaustive list of these properties follows, while other properties are specific to certain field types.

<code>type</code>	The column type, which corresponds indirectly to an SQL column type. See the table of common field types below.
<code>vname</code>	A reference to a language string in either the module or application language files representing a label for this field.
<code>vname_list</code>	A language string to override <code>vname</code> in the context of list column labels.
<code>audited</code>	A flag indicating that updates to this field are logged to the associated audit table, as long as <code>audit_enabled</code> is set in the model detail descriptor.
<code>comment</code>	A string describing the usage of this field.
<code>default</code>	A default value for the column when none is specified by the user.
<code>editable</code>	Set to false to disable user editing of a field, including on new records.
<code>importable</code>	Generally defaulting to true, set this flag to false to hide this field inside the ImportDB module.
<code>len</code>	The length of the corresponding database column in characters.
<code>massupdate</code>	A flag to control visibility of this field on the ListView's mass-update panel.
<code>reportable</code>	Whether to allow make this field available for reports.
<code>required</code>	Marks this field as required, meaning it must contain a non-null value.
<code>updateable</code>	Like <code>editable</code> , disables user updates to the field, but only for existing records.
<code>unified_search</code>	A flag indicating that this field should be added to the default unified search filter.

### System-Level Field Descriptors

These field descriptors may be referenced to include standard field descriptors within a model descriptor file. Properties of the standard field descriptors may be overridden by listing them underneath this key.

<code>app.id</code>	
<code>app.date_entered</code>	

app.date_modified	
app.modified_user	
app.assigned_user	
app.created_by_user	
app.currency	
app.exchange_rate	

### Common Field Types

id	A 36-character string field containing a unique, system-generated identifier.
varchar	A string value.
tinyint, int, float, double, currency	Standard numeric field types.
bool	A true or false value, usually represented as an SQL <code>tinyint</code> . Fields of this type are rendered as checkboxes.
date, time, datetime	Standard date and time field types.
duration	A duration field, stored as an integer representing a number of a minutes.
enum	A dropdown list, usually represented as a <code>varchar</code> column and having an associated <code>options</code> array.
multienum	A set of values selected from a dropdown list.
ref	<p>Representing a reference to a record in another model. This field does not map to a database column itself, but will have an associated ID field (automatically created, or named by the <code>id_name</code> property). When this field is queried by adding it to a form or list layout, a link to the related record is rendered using the target's display name. Normally a <code>ref</code> field defines <code>bean_name</code>, representing the name of the target model.</p> <p>Otherwise, a <code>ref</code> field must define <code>dynamic_module</code> (a column name), in which case it can target a record in one of multiple modules. See the Calls or Tasks modules for examples of this usage.</p>
text	This field type represents a multi-line text field and is stored in an SQL <code>text</code> column.
html	An HTML field, such as the body of an email template.

## Table Indexes

For improved speed in performing common searches, multiple indexes may be defined on each model descriptor. These are contained within the `indices` section. Each entry consists of an array key representing the unique name for the index, along with an array of properties. For most purposes the only relevant property is `fields`, containing an array of column names used to construct the index.

### *A sample index definition used by the EmailTemplate model*

```
indices
  idx_email_template_name
    fields
      - name
```

## Model Links and Relationships

Model link definitions are used to manage one-to-many and many-to-many associations between records, while one-to-one or many-to-one record linkages are generally represented using `ref` fields. These link definitions are most often used as the basis for sub-panels, and are contained in the `links` section of the model descriptor file.

### *Sample link definitions used by the Account model*

```
links
  members
    relationship: member_accounts
    module: Accounts
    bean_name: Account
    vname: LBL_MEMBERS
  tasks
    relationship: account_tasks
    module: Tasks
    bean_name: Task
    vname: LBL_TASKS
```

Each link must reference a corresponding relationship, which may be defined in the current model descriptor file or elsewhere. When defined inside a `bean` descriptor file relationship definitions resemble the following (corresponding to the link definitions above).

*Sample relationship definitions used by the Account model*

```
relationships
  member_accounts
    relationship_type: one-to-many
    key: parent_id
    target_bean: Account
    target_key: id
  account_tasks
    relationship_type: one-to-many
    key: id
    target_bean: Task
    target_key: parent_id
    role_column: parent_type
    role_value: Accounts
```

In the above relationship descriptors, the `key` property names a field in the current model definition used to establish the relationship. Matching records in the table defined by the `target_bean` model are found by equating its `target_key` field to the value of `key`.

Relationship descriptors may also define a `role_column` and `role_value` to further restrict the targeted set of records. This is generally used when the referenced field is a `ref` field with `dynamic_module` defined.

Relationships defined within `link` model descriptors have slightly different formatting, as seen below. Note that the relationship shares the name of the `link` model in this case.

*A sample relationship definition used by the discounts\_products link model*

```
relationships
  discounts_products
    relationship_type: many-to-many
    lhs_key: id
    lhs_bean: Product
    join_key_lhs: product_id
    rhs_key: id
    rhs_bean: Discount
    join_key_rhs: discount_id
```

In this definition, `lhs` represents the (arbitrary) left-hand side of the relationship and `rhs` the right. `join_key_lhs` and `join_key_rhs` are fields defined by this link model, while `lhs_key` is a field in the `lhs_bean` model, and `rhs_key` is a field in the `rhs_bean` model. You can think of the SQL join statement as setting `lhs_bean.lhs_key = join_key_lhs` and `join_key_rhs = rhs_bean.rhs_key`.

### ➔ 9.3.5 Localization

In **info@hand 7**, the organization of translatable language strings has changed. The `language/` subdirectory of each module directory is expected to contain at least two files: `lang.en_us.meta.php` and `lang.en_us.strings.php`. The first contains the label for this module (the `label` key in the excerpt below), which is automatically collected in the system-wide `$app_strings['moduleList']` array familiar from previous **info@hand** versions. This file may also define a module from which to inherit language strings (`inherit_from`) – useful in the case of similar modules which share common strings. This functionality can help to reduce the translation work required and is also supported by the javascript framework.

```
lang.en_us.meta.php from the Invoice module

detail
  label: Invoices
  comment: en_us language file for Invoice module
  inherit_from: Quotes
```

Module language strings are listed in the file `lang.en_us.strings.php`. This is a simple array of key-value pairs, and should not contain any nested arrays. These strings may be referenced in field descriptors, in layouts descriptors, and may be accessed programmatically using the system function `translate($label, $module)`. If module-specific language arrays are to be used, they may be stored in `lang.en_us.lists.php`.

### ➔ 9.3.6 Model Display Descriptors

...

### ➔ 9.3.7 Layout Descriptors

The use of HTML templates in **info@hand 7** is strongly discouraged in favour of the new form generation system. `DetailView` and `EditView` forms are now rendered by the `StandardDetailManager` class (in `include/DetailView`). `ListViews` are rendered by the `ListViewManager` and `ListFormatter` classes (in `include/ListView`). The layout templates for all actions are located in the `views/` module subdirectory and prefixed with the relevant action name.

<code>view.Standard.php</code>	The <code>DetailView</code> form layout. Other layouts named as <code>view.*.php</code> may be accessed using specific values for the <code>layout</code> request parameter (particularly when using a tabbed form layout).
<code>edit.Standard.php</code>	The standard <code>EditView</code> form layout.
<code>list.Standard.php</code>	The standard <code>ListView</code> column layout. Other layouts named as <code>list.*.php</code> may be made accessible by listing them in the Model Display Metadata.

popup.Standard.php	The standard layout for a Popup ListView (shown for example when the popup button on a ref input field is used). If not present then list.Standard.php is used instead.
subpanel.Standard.php	The sub-panel layout used for this module. If not present, then list.Standard.php will be used to generate the sub-panel instead.
search.Standard.php	The search form layout used on the 'Quick Filter' module ListView.
additional.Standard.php	The DetailView-style form layout used in the 'additional details' popup generated on various ListViews.

Each layout descriptor begins with a `detail` array defining the layout type (which should generally equal the prefix on the filename). Certain layouts including `view` and `edit` may also define a `title`, representing a default title to be used at the top of the form. Further metadata may also be contained in this header. Following this is the `layout` array, which contains the details of the form layout.

Layout descriptors can be grouped into two basic formats. The `list`, `popup` and `subpanel` layouts define a `columns` array underneath `layout`, containing an ordered list of column descriptors. A column descriptor may consist of a string referencing a field in the model, or an array. If an array, that array should generally define a `field` key, again referencing a field in the model. Using an array also allows the customization of properties like `width` (an integer representing the column width in characters) and `vname` (an alternate column label). Array column descriptors may also define `add_fields`, another array of field names to be added on subsequent lines within each column entry.

```

A sample module list layout with two columns
detail
  type: list
layout
  columns
    --
    field: name
    add_fields: [type]
    width: 60
    - assigned_user
    
```

The layout descriptors for `view` and `edit` layouts follow a separate common format. The primary entry within the `layout` array is `sections`, which defines a list of top-level form sections.

Each `sections` entry is an array. Start by defining a unique `id` for the section. This may be used in javascript to obtain a reference to the containing element. Next, the `vname` (a header for the section) may be provided. For a `view` or `edit` layout, the default number of layout columns is 2, but this may

be overridden by setting the `columns` attribute. For `search` layouts an appropriate number of columns is normally decided based on the number of fields to be rendered.

Field references are then provided in the `elements` array within the `sections` entry. When the form is rendered, these are generally presented as a pair of table cells, one for the label and one for the representation of the field (which will vary depending on whether the field is editable). Each entry in `elements` may be either a string, for a simple field reference, or an array for more complicated cases. If an array is used then various properties may be overridden, including the `colspan` for this field, the `vname` (field label), and some field type-specific properties. Setting a custom value for the `colspan` is demonstrated by the `description` field in the sample code below.

### *A sample module view (DetailView) layout*

```
detail
  type: view
  title: LBL_MODULE_TITLE
  layout
    sections
      --
      id: main
      elements
        - name
        - type
        -
        - date_modified
        - assigned_user
        - date_entered
        --
        name: description
        colspan: 2
    subpanels
      - accounts
      - contacts
```

For `view` layouts, it often makes sense to define a list of sub-panels following the form sections. These are entered in the `subpanels` array, a child of `layout`. Each entry here references an entry in the `links` section of the model descriptor (see Model Links and Relationships).

In both `view` and `edit` layouts it is also possible to define custom form buttons. These are entered in the `form_buttons` array, also child of `layout`. Each entry should have a unique key representing the name of the button. It should define a `vname` (button label), may define a custom button `icon`, and can specify `async: false` if the default behaviour of performing a partial page load is not desired. The `params` attribute defines a list of properties to be overridden in the resulting HTTP request. If more complex behaviour is required, a custom javascript handler may be provided in an `onclick` attribute.

*Defining a custom form button*

```
# ...
layout
  form_buttons
    pdf
      vname: LBL_PDF_BUTTON_LABEL
      icon: icon-print
      params
        action: PDF
      async: false
  sections
    # ...
```

Because the classic DetailView.php, EditView.php, Save.php and Delete.php files are no longer present, custom behaviours when displaying, creating and updating records should be specified within model hooks. See the section on Business Logic Hooks for more information.