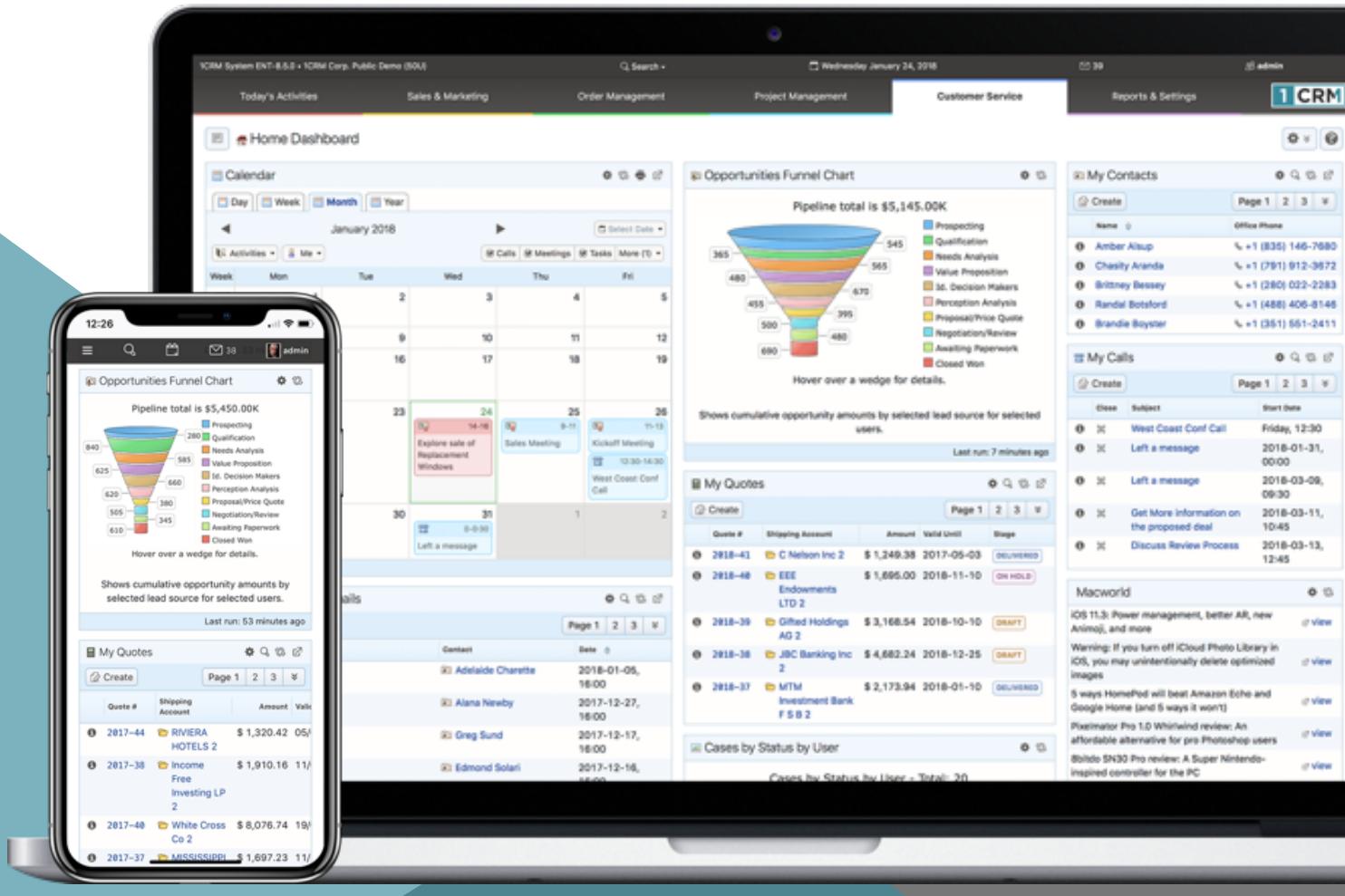
1CRM

1CRM DEVELOPER GUIDE

A Comprehensive Guide to Developing Customizations and Extensions for 1CRM



Navigating This Guide:

This Guide has been designed to be helpful both as a printed document, and as an electronic document accessed on your computer screen. If you are accessing it via a PDF viewer such as Acrobat Reader or Mac Preview, please notice:

1. The Table of Contents entries are all live hyperlinked to the pages to which they refer.
2. At the top of each page there is a link back to the start of the Table of Contents.
3. Be sure to use the search feature of your PDF reader.

Version 8.6, February 2021. This document is subject to change without notice.

Table of Contents

1.0 Welcome

Thank you for using 1CRM! Release 8.6 of the 1CRM Customer Relationship and Business Management (CRBM) System is designed to further energise your organisation's efforts to efficiently organise and maintain information that is crucial to many aspects of your business. 1CRM enables organizations to do business, better.

The 1CRM system is available in four Editions:

- Startup Edition: Free for use On Premise. Request a license key and download link, and you can install Startup Edition on your own server at no charge. It offers all the features of 1CRM Professional Edition, but is limited to 3 Users, 300 Accounts, 750 Leads, 750 Contacts, and 750 Targets. Although it only offers Community support, and no updates, the Startup Edition is a great way for an early stage business to get itself organized and productive while operating on a shoestring budget!
- Startup+ Edition: Similar to the Startup Edition, but for somewhat larger firms, with capacity limits of 10 users, 600 Accounts, 1,500 Contacts, 1,500 Leads and 1,500 Targets. Unlike the Startup Edition, this is a commercial product, available on the 1CRM Cloud or for On Premise software installation.
- Professional Edition: Formerly known as **info@hand**, 1CRM Professional is our mainstream small business CRM product offering, available on the 1CRM Cloud or for On Premise software installation.
- Enterprise Edition: Our premium product. It offers all the features of 1CRM Professional Edition, plus a number of additional features of particular interest to larger, more sophisticated businesses. Administrators can use the Module Designer and PDF Form Designer to create more advanced customizations. Price Books let you establish pricing for multiple client levels. The iOS client provides optimized system access from an iPhone. And Advanced Reporting offers more sophisticated reporting capabilities.

Unlike most CRM solutions, 1CRM offers comprehensive Order Management. It includes a Product Catalog, plus the ability to create Quotations, Sales Orders and Invoices using products from the Catalog. Incoming Payments may be received and allocated against invoices, and the system can produce PDF documents for Quotes, Sales Orders, Invoices, Receipts, and Statements. Purchase Orders may also be created, and Outgoing Payments recorded against them.

1CRM also offers extensive features for Project Management, Service Management, and general office administration (including Expense Reports, Timesheets, Vacation scheduling and tracking, and HR).

Most importantly, the 1CRM system seamlessly blends all of these capabilities into an intuitive and friendly interface. The instructions in this guide will introduce you to the most important CRM concepts and help you get familiar with using your 1CRM system.

1.1 About this Guide

This guide is written for those individuals tasked with adapting the 1CRM system for specialized uses. It is current with the details of operation for 1CRM 8.6. It is designed to explain methods for customization of the 1CRM system, maintaining compatibility with future upgrades to the base product as much as possible.

Readers are expected to be proficient in software development in a web-based environment, including a working knowledge of Apache, PHP and MySQL. For user interface enhancements, capability in JavaScript and CSS may be necessary.

1.2 Who Should Read this Guide?

This 1CRM *Developer Guide* is intended for IT personnel and contractors who are developing custom extensions for the 1CRM system. It is also meant for project managers who need to estimate the scope and duration of development work.

It is not intended for conventional users who wish to record and track company activities and outcomes, or for system administrators looking to install and optimize the 1CRM system – those topics are dealt with in the 1CRM *User Guide* and *Implementation Guide*.

1.3 Additional Documentation

The 1CRM Customer Relationship and Business Management (CRBM) system offers extensive documentation for the installation and use of its various components. Click on any link below to download that document, or click here to see all our 1CRM documentation:

- User Guide
- Mobile User Guide
- Implementation Guide
- Developer Guide

2.0 Integration with Web Services

2.1 Compatibility APIs

A number of our clients have been interested to use a variety of SugarCRM add-on products from third-party vendors, since the 1CRM 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 1CRM 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 6.4. Note that the methods available (and reported in the generated WSDL file) will depend on the entry point used: soap.php for the evolving, native SOAP API, and service/v[#]/soap.php for specific SugarCRM API versions. ***Note:*** The API will be deprecated soon, and removed from the 1CRM codebase.

When a third party software uses a SOAP or REST call to 1CRM to ask for the version of SugarCRM software, 1CRM replies with this version info (6.4) by default. If you wish for some reason to change this answer, you may do so, by overriding the *soap.public_version* setting in your *local_config.php* file. This ability to override 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 6.4 solely by means of the SOAP or REST API, then there is a very good chance it will also work just fine with 1CRM, although there are some changes to the database structure of base modules which can lead to incompatibilities.

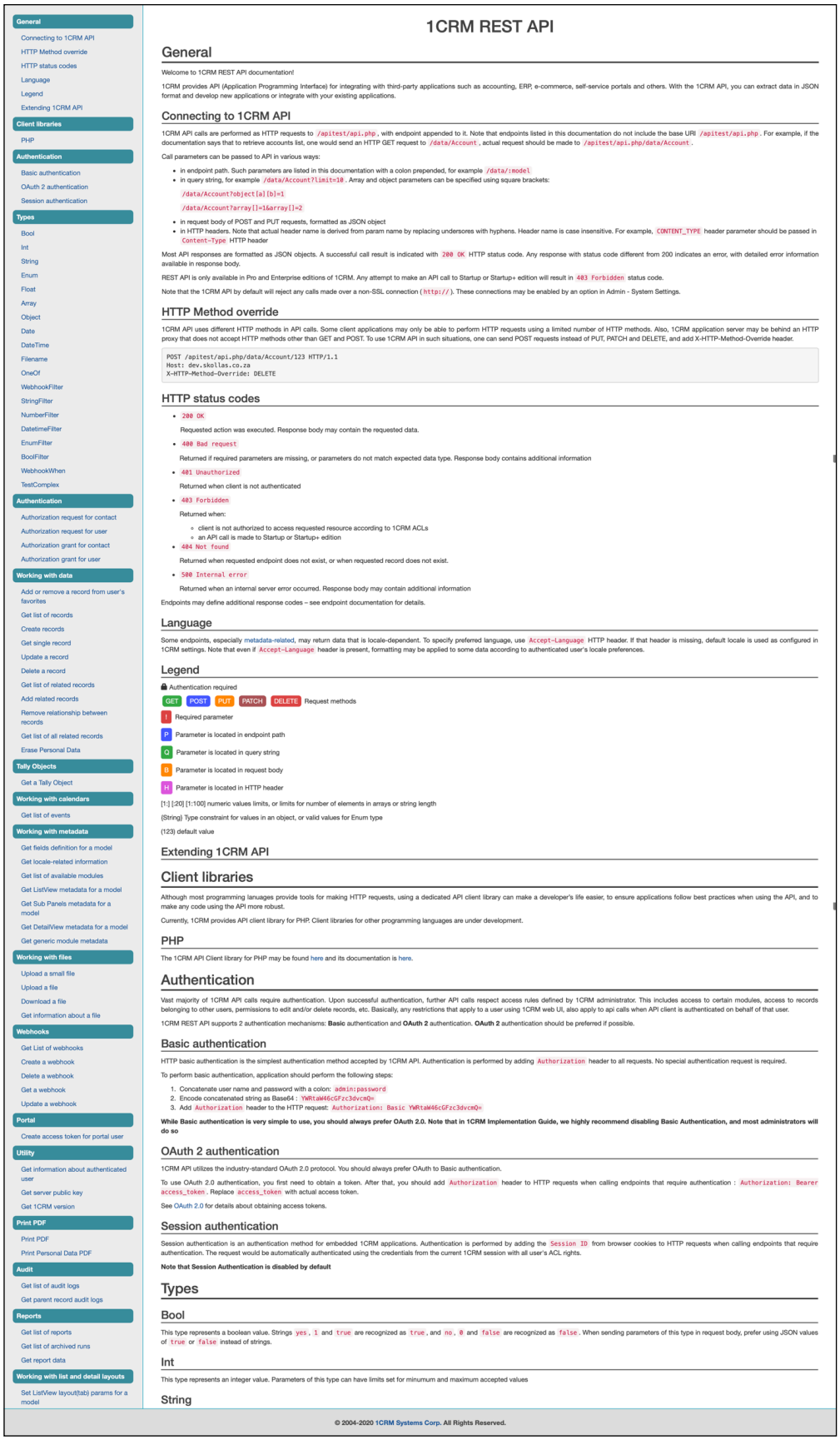
If you want to write your own software which accesses 1CRM via the SugarCRM SOAP API, you should follow the SugarCRM documentation found here. But if you have any choice, we do not recommend it. Use the vastly more capable *Next Generation REST API* described below.

2.2 Next Generation REST API

Beginning with System 8, 1CRM offers an all-new REST API. It may be accessed at `/api.php` from your URL root. You can also view and navigate the API documentation there if you browse to that URL, as shown below.

Note: While this Developer Guide API content is updated regularly, the live API documentation at `/api.php` from your URL root should always be used as the definitive reference information.

Note: This API is used by the 1CRM Mobile iPhone app. By default your 1CRM system is configured only to allow the app to communicate with it securely, via SSL using a URL beginning with `https://`. All 1CRM Cloud installations have an SSL certificate installed by default, so you can simply enter your URL as you normally would, but with `https://` at the front (which you may or may not normally do in your browser anyway). If your 1CRM On Premise instance does not have an SSL certificate, you can allow non-SSL use of the API by going to the *Admin - API and OAuth Settings* screen, and turn on the option *Allow API Calls via insecure connections (http://)*.



1CRM REST API

General

Welcome to 1CRM REST API documentation!

1CRM provides API (Application Programming Interface) for integrating with third-party applications such as accounting, ERP, e-commerce, self-service portals and others. With the 1CRM API, you can extract data in JSON format and develop new applications or integrate with your existing applications.

Connecting to 1CRM API

1CRM API calls are performed as HTTP requests to `/apitest/api.php`, with endpoint appended to it. Note that endpoints listed in this documentation do not include the base URI `/apitest/api.php`. For example, if the documentation says that to retrieve accounts list, one would send an HTTP GET request to `/data/Account`, actual request should be made to `/apitest/api.php/data/Account`.

Call parameters can be passed to API in various ways:

- in endpoint path. Such parameters are listed in this documentation with a colon prepended, for example `/data/:model`
- in query string, for example `/data/Account?limit=10`. Array and object parameters can be specified using square brackets:

  `/data/Account?object[a][b]=1`

  `/data/Account?array[]=1&array[]=2`
- in request body of POST and PUT requests, formatted as JSON object
- in HTTP headers. Note that actual header name is derived from param name by replacing undersores with hyphens. Header name is case insensitive. For example, `CONTENT_TYPE` header parameter should be passed in `Content-Type` HTTP header

Most API responses are formatted as JSON objects. A successful call result is indicated with `200 OK` HTTP status code. Any response with status code different from 200 indicates an error, with detailed error information available in response body.

REST API is only available in Pro and Enterprise editions of 1CRM. Any attempt to make an API call to Startup or Startup+ edition will result in `403 Forbidden` status code.

Note that the 1CRM API by default will reject any calls made over a non-SSL connection (`http://`). These connections may be enabled by an option in Admin - System Settings.

HTTP Method override

1CRM API uses different HTTP methods in API calls. Some client applications may only be able to perform HTTP requests using a limited number of HTTP methods. Also, 1CRM application server may be behind an HTTP proxy that does not accept HTTP methods other than GET and POST. To use 1CRM API in such situations, one can send POST requests instead of PUT, PATCH and DELETE, and add X-HTTP-Method-Override header.

```
POST /apitest/api.php/data/Account/123 HTTP/1.1
Host: dev.skollas.co.za
X-HTTP-Method-Override: DELETE
```

HTTP status codes

- `200 OK`

  Requested action was executed. Response body may contain the requested data.
- `400 Bad request`

  Returned if required parameters are missing, or parameters do not match expected data type. Response body contains additional information
- `401 Unauthorized`

  Returned when client is not authenticated
- `403 Forbidden`

  Returned when:
  - client is not authorized to access requested resource according to 1CRM ACLs
  - an API call is made to Startup or Startup+ edition
- `404 Not found`

  Returned when requested endpoint does not exist, or when requested record does not exist.
- `500 Internal error`

  Returned when an internal server error occurred. Response body may contain additional information

Endpoints may define additional response codes – see endpoint documentation for details.

Language

Some endpoints, especially metadata-related, may return data that is locale-dependent. To specify preferred language, use `Accept-Language` HTTP header. If that header is missing, default locale is used as configured in 1CRM settings. Note that even if `Accept-Language` header is present, formatting may be applied to some data according to authenticated user's locale preferences.

Legend

Authentication required

GET POST PUT PATCH DELETE Request methods

! Required parameter

P Parameter is located in endpoint path

Q Parameter is located in query string

B Parameter is located in request body

H Parameter is located in HTTP header

[1:] [:20] [1:100] numeric values limits, or limits for number of elements in arrays or string length

{String} Type constraint for values in an object, or valid values for Enum type

(123) default value

Extending 1CRM API

Client libraries

Although most programming lanuages provide tools for making HTTP requests, using a dedicated API client library can make a developer's life easier, to ensure applications follow best practices when using the API, and to make any code using the API more robust.

Currently, 1CRM provides API client library for PHP. Client libraries for other programming languages are under development.

PHP

The 1CRM API Client library for PHP may be found here and its documentation is here.

Authentication

Vast majority of 1CRM API calls require authentication. Upon successful authentication, further API calls respect access rules defined by 1CRM administrator. This includes access to certain modules, access to records belonging to other users, permissions to edit and/or delete records, etc. Basically, any restrictions that apply to a user using 1CRM web UI, also apply to api calls when API client is authenticated on behalf of that user.

1CRM REST API supports 2 authentication mechanisms: **Basic** authentication and **OAuth 2** authentication. **OAuth 2** authentication should be preferred if possible.

Basic authentication

HTTP basic authentication is the simplest authentication method accepted by 1CRM API. Authentication is performed by adding `Authorization` header to all requests. No special authentication request is required.

To perform basic authentication, application should perform the following steps:

1. Concatenate user name and password with a colon: `admin:password`
2. Encode concatenated string as Base64 : `YWRtaW46cGFzc3dvcmQ=`
3. Add `Authorization` header to the HTTP request: `Authorization: Basic YWRtaW46cGFzc3dvcmQ=`

While Basic authentication is very simple to use, you should always prefer OAuth 2.0. Note that in 1CRM Implementation Guide, we highly recommend disabling Basic Authentication, and most administrators will do so

OAuth 2 authentication

1CRM API utilizes the industry-standard OAuth 2.0 protocol. You should always prefer OAuth to Basic authentication.

To use OAuth 2.0 authentication, you first need to obtain a token. After that, you should add `Authorization` header to HTTP requests when calling endpoints that require authentication : `Authorization: Bearer access_token`. Replace `access_token` with actual access token.

See OAuth 2.0 for details about obtaining access tokens.

Session authentication

Session authentication is an authentication method for embedded 1CRM applications. Authentication is performed by adding the `Session ID` from browser cookies to HTTP requests when calling endpoints that require authentication. The request would be automatically authenticated using the credentials from the current 1CRM session with all user's ACL rights.

Note that Session Authentication is disabled by default

Types

Bool

This type represents a boolean value. Strings `yes`, `1` and `true` are recognized as `true`, and `no`, `0` and `false` are recognized as `false`. When sending parameters of this type in request body, prefer using JSON values of `true` or `false` instead of strings.

Int

This type represents an integer value. Parameters of this type can have limits set for minumum and maximum accepted values

String

2.2.1 General

1CRM provides an API (Application Programming Interface) for integrating with third-party applications such as accounting, ERP, e-commerce, self-service portals and others. With the 1CRM API, you can extract data in JSON format and develop new applications or integrate with existing applications.

Connecting to the 1CRM API

1CRM API calls are performed as HTTP requests to `/1crm/api.php`, with endpoint appended to it. Note that endpoints listed in this documentation do not include the base URI `/1crm/api.php`. For example, if the documentation says that to retrieve accounts list, one would send an HTTP GET request to `/data/Account`, actual request should be made to `/1crm/api.php/data/Account`.

Call parameters can be passed to API in various ways:

- in endpoint path. Such parameters are listed in this documentation with a colon prepended, for example `/data/:model`
- in query string, for example `/data/Account?limit=10`. Array and object parameters can be specified using square brackets:
  `/data/Account?object[a][b]=1`
  `/data/Account?array[]=1&array[]=2`
- in request body of POST and PUT requests, formatted as JSON object
- in HTTP headers. Note that actual header name is derived from param name by replacing underscores with hyphens. Header name is case insensitive. For example, `CONTENT_TYPE` header parameter should be passed in `Content-Type` HTTP header

Most API responses are formatted as JSON objects. A successful call result is indicated with `200 OK` HTTP status code. Any response with status code different from 200 indicates an error, with detailed error information available in response body.

REST API is only available in Pro and Enterprise editions of 1CRM. Any attempt to make an API call to Startup or Startup+ edition will result in `403 Forbidden` status code.

Note that the 1CRM API by default will reject any calls made over a non-SSL connection (`http://`). These connections may be enabled by an option in Admin - System Settings.

HTTP Method override

1CRM API uses different HTTP methods in API calls. Some client applications may only be able to perform HTTP requests using a limited number of HTTP methods. Also, 1CRM application server may be behind an HTTP proxy that does not accept HTTP methods other than GET and POST. To use 1CRM API in such situations, one can send POST requests instead of PUT, PATCH and DELETE, and add X-HTTP-Method-Override header.

```
POST /api.php/Account/123 HTTP/1.1

Host: 1crm.ca

X-HTTP-Method-Override: DELETE
```

HTTP status codes

`200 OK`
Requested action was executed. Response body may contain the requested data.

`400 Bad request`
Returned if required parameters are missing, or parameters do not match expected data type. Response body contains additional information

`401 Unauthorized`
Returned when client is not authenticated.

`403 Forbidden`
Returned when:

- client is not authorized to access requested resource according to 1CRM ACLs
- an API call is made to Startup or Startup+ edition

`404 Not found`
Returned when requested endpoint does not exist, or when requested record does not exist.

`500 Internal error`
Returned when an internal server error occurred. Response body may contain additional information

Endpoints may define additional response codes – see endpoint documentation for details.

Language

Some endpoints, especially metadata-related, may return data that is locale-dependent. To specify preferred language, use `Accept-Language` HTTP header. If that header is missing, default locale is used as configured in 1CRM settings. Note that even if `Accept-Language` header is present, formatting may be applied to some data according to authenticated user's locale preferences.

Legend

Authentication required

GET POST PUT PATCH DELETE Request methods

! Required parameter

P Parameter is located in endpoint path

Q Parameter is located in query string

B Parameter is located in request body

H Parameter is located in HTTP header

[1:] [:20] [1:100] numeric values limits, or limits for number of elements in arrays or string length

{String} Type constraint for values in an object

(123) default value

Extending 1CRM API

This article describes how developers of 1CRM extension modules can extend the API. Let's suppose that you created a custom module named `Echo`.

Registering an API group

Each endpoint in the 1CRM API belongs to a group. Grouping does not affect the API functionality, but it is useful for API documentation formatting. We start by creating a file named `modules/Echo/ext/config/standard/api_config.php`:

```
groups
    myutils
        title: "My Utils"
        docs: "Miscellanous useful utilities"
```

Registering endpoints

Next step is to register an endpoint.

```
groups
    myutils
        title: My Utils
        docs: "Miscellanous useful utilities"
endpoints
    --
        file: modules/Echo/API.php
        class: EchoEndpoint
        group: myutils
```

This new config file section adds an endpoint to `myutils` group. The endpoint will be defined in `EchoEndpoint` class, located in `modules/Echo/API.php` file.

Endpoint class

Endpoint class must extend `OneCRM\API\Endpoint`:

```
<?php
use OneCRM\API\Endpoint;
class EchoEndpoint extends Endpoint {
}
```

Required methods

First, we need to define several required methods:

```php
<?php

use OneCRM\API\Endpoint;

class EchoEndpoint extends Endpoint {

    public function allowedMethods() {
        return ['post'];
    }

    public function isAuthEndpoint() {
        return false;
    }

    public function requiredScopes($method) {
        return ['read'];
    }

    public function descriptor($method) {
        if ($method == 'post')
            return $this->descriptorPost();
    }

    public function endpoint() {
        return '/ext/echo';
    }

}
```

This endpoint uses `/ext/echo` path. Each endpoint can support multiple HTTP request methods (GET,POST, PUT, etc.), so we have to return all method it supports from allowedMethods(). In this case, the only supported HTTP method is POST.

Authentication scope required by this endpoint is `read`. You may also specify `write` and `profile` if needed.

`isAuthEndpoint()` should always return false.

Endpoint descriptor

`descriptor()` method must return endpoint descriptor describing endpoint parameters and returned value. In our case, it will return a descriptor only for HTTP POST method, because that is the only one supported.

We start with a minimal descriptor containing only a title and documentation. Notice that here we reference documentation placed into a separate file — `modules/Echo/docs/docs.md`.

```php
<?php

use OneCRM\API\Endpoint;

class EchoEndpoint extends Endpoint {
    // ..... skipped ....

    private function descriptorPost() {
        return [
            'title' => 'Echo',
            'docs' => '@+modules/Echo/docs/
docs', // docs in a .md file, notice that extension is omited
        ];
    }
}
```

Parameters

An endpoint can accept a number of parameters that can be passed when making an API call.

```php
<?php

use OneCRM\API\Endpoint;

class EchoEndpoint extends Endpoint {
    // ..... skipped ....
    private function descriptorPost() {
        return [
            'title' => 'Echo',
            'docs' => '@+modules/Echo/docs/
docs', // docs in a .md file, notice that extension is omited
            'params' => [
                'data' => [
                    'location' => 'body',
                    'type' => 'Object',
                    'docs' => 'Data to echo',
                    'required' => true,
                ],
                'format' => [
                    'location' => 'query',
                    'type' => 'Enum',
                    'options' => ['json', 'xml'],
                    'docs' => 'Output format',
                    'required' => false,
                ],
            ],
            'return' => [
                '' => [
                    'type' => 'String',
                    'docs' => 'Data echoed back',
                    'required' => true,
                ],
            ],
        ];
    }
}
```

Here we declared a required parameter `data` which must be passed in request body, and an optional parameter `format` which can be passed in query string. These parameters will be checked when serving API calls, and an error will be returned if passed parameters do not match the specification.

You can also describe the data returned by the endpoint, but this serves for documentation purposes only.

HTTP handler

Now it is time to create a method that will process requests and return responses. The method name must match the corresponding HTTP method.

```
<?php

use OneCRM\API\Endpoint;

class EchoEndpoint extends Endpoint {
    // ..... skipped ....
    protected function post($input, $res, $req) {

        if ($input['format'] != 'xml')
            return $res->withJSON($input['data']);

        $body = $res->getBody();
        $body->rewind();

        $xml = new SimpleXMLElement('<echo/>');
        $body->write($this->arrayToXML($input['data'], $xml, 'element'));

        return $res->withHeader('Content-type', 'text/xml; charset=UTF-8')-
>withBody($body);
    }

    public function arrayToXML($array, SimpleXMLElement $xml, $child_name)
    {
        foreach ($array as $k => $v) {
            if(is_array($v)) {
                (is_int($k)) ? $this->arrayToXML($v, $xml-
>addChild($child_name), $v) : $this->arrayToXML($v, $xml-
>addChild(strtolower($k)), $child_name);
            } else {
                (is_int($k)) ? $xml->addChild($child_name, $v) : $xml-
>addChild(strtolower($k), $v);
            }
        }

        return $xml->asXML();
    }

}
```

This is just an example, normally you want to always use `withJSON()`.

1CRM API uses the Slim framework. Refer to its documentation to learn how to work with Request and Response objects.

Returning errors

If during a request processing an error occurs, you can throw one of `BadRequest`, `InternalError`, `NotFound` from `OneCRM\API\Errors` namespace.

```
<?php

use OneCRM\API\Endpoint;
use OneCRM\API\Errors;

class EchoEndpoint extends Endpoint {
    // ..... skipped ....
    protected function post($input, $res, $req) {

        if (!empty($input['data']['prohibited']))
            throw new Errors\BadRequest("prohibited parameter passed");
        if ($input['format'] != 'xml')
            return $res->withJSON($input['data']);

        $body = $res->getBody();
        $body->rewind();

        $xml = new SimpleXMLElement('<echo/>');
        $body->write($this->arrayToXML($input['data'], $xml, 'element'));

        return $res->withHeader('Content-type', 'text/xml; charset=UTF-8')-
>withBody($body);
    }
    // ..... skipped ....
}
```

Client libraries

Although most programming languages provide tools for making HTTP requests, using a dedicated API client library can make a developer's life easier, to ensure applications follow best practices when using the API, and to make any code using the API more robust.

Currently, 1CRM provides API client library for PHP. Client libraries for other programming languages are under development.

PHP

The 1CRM API Client library for PHP may be found here and its documentation is here.

Authentication

The vast majority of 1CRM API calls require authentication. Upon successful authentication, further API calls respect access rules defined by 1CRM administrator. This includes access to certain modules, access to records belonging to other users, permissions to edit and/or delete records, etc. Basically, any restrictions that apply to a user using 1CRM web UI, also apply to api calls when API client is authenticated on behalf of that user.

The 1CRM REST API supports 2 authentication mechanisms: **Basic** authentication and **OAuth 2** authentication. **OAuth 2** authentication should be preferred if possible.

Basic authentication

HTTP basic authentication is the simplest authentication method accepted by 1CRM API. Authentication is performed by adding `Authorization` header to all requests. No special authentication request is required.

To perform basic authentication, application should perform the following steps:

1. Make MD5 hash of password, for example `supersecret` becomes `9a618248b64db62d15b300a07b00580b`

2. Concatenate user name and password hash with a colon: `admin:9a618248b64db62d15b300a07b00580b`

3. Encode concatenated string as Base64 : `YWRtaW46OWE2MTgyNDhiNjRkYjYyZDE1YjMwMGEwN2IwMDU4MGI=`

4. Add `Authorization` header to the HTTP request: `Authorization: Basic YWRtaW46OWE2MTgyNDhiNjRkYjYyZDE1YjMwMGEwN2IwMDU4MGI=`

While Basic authentication is very simple to use, you should always prefer OAuth 2.0. Note that in the 1CRM Implementation Guide, we highly recommend disabling Basic Authentication, and most administrators should do so.

OAuth 2 authentication

1CRM API utilizes the industry-standard OAuth 2.0 protocol. You should always prefer OAuth to Basic authentication.

To use OAuth 2.0 authentication, you first need to obtain a token. After that, you should add `Authorization` header to HTTP requests when calling endpoints that require authentication : `Authorization: Bearer access_token`. Replace `access_token` with actual access token.

Application registration

In order to make calls to the 1CRM API using OAuth 2.0, you need to register an application, or API client. Registered clients are assigned a unique Client ID (client_id) and a unique Client Secret (client_secret). Make sure to store the Client Secret securely.

1CRM supports 2 types of API clients: public and private.

Private API clients can be registered by 1CRM administrator, and are intended for use by your organization only. Public API clients are managed by 1CRM Systems Corp., and represent 3rd party applications. If you are developing an application that can be potentially useful for all 1CRM customers, you should apply for a public API client.

Client ID and Client Secret for private clients are available from 1CRM admin interface. For public clients, 1CRM will provide Client ID and Client Secret after creating your API Client.

Resource owners

One of the roles defined by OAuth 2.0 Authorization Framework is Resource Owner:

> *An entity capable of granting access to a protected resource. When the resource owner is a person, it is referred to as an end-user.*

Naturally, in 1CRM, its users are resource owners. API client can be granted access on behalf of a 1CRM user. But there are many cases, when it is also desirable to allow Contacts to authenticate themselves: using 1CRM as single sign-on server, customer portals, etc.

1CRM defines two types of Resource Owners: User and Contact. Depending on type of resource owner you want to authenticate, you use different endpoints. Note that Contact access token only provides access to Contact's own information such as name, email address and telephone number.

Scopes

When requesting access to protected resources, API client can specify the scope of request. 1CRM supports 3 scopes:

- `read` client requests access to read information from 1CRM
- `write` client requests access to write information to 1CRM
- `profile` client requests access to Resource Owner's account information

For Resource Owner of Contact type, only `profile` scope is allowed. `read` and `write` scopes are subject to limitations set by 1CRM administrator using 1CRM ACL system.

If you do not specify a scope in authorization request, `profile` will be used by default.

Authentication workflow

1CRM allows you to use OAuth 2.0 via one of five flows:

- Authorization Code Grant - use this flow if you want to authorize a web application.
- Implicit Grant - use this flow if you want to authorize a user-agent, a desktop or a mobile application.
- Resource Owner Password Credentials Grant - use this flow if you want to authenticate directly via 1CRM API using resource owner's password credentials.
- Client Credentials Grant - use this flow to authorize an application on behalf of predefined 1CRM user. The actual user that gains access in this flow is decided by 1CRM administrator.
- Refresh Token Grant - use this flow to renew an access token when the current one expires.

Authorization Code grant

Authorization Code grant is most suitable for web server applications or native mobile applications. The flow should be very familiar if you have ever signed into a web app using your Facebook or Google account.

Step one

The client will redirect the user's web browser to `/auth/user/authorize` or `/auth/contact/authorize` with the following parameters in the query string:

- `response_type` with the value `code`
- `client_id` with the client identifier
- `redirect_uri` with the client redirect URI. This parameter is optional, and defaults to first URI from the list configured by 1CRM administrator. If specified, this parameter must match one of redirect URIs configured by 1CRM administrator.
- `state` optional CSRF token. This parameter is optional but highly recommended. You should store the value of the CSRF token in the user's session to be validated when they return to your application

The user will be asked to login and approve the client. If the user approves the client, they will be redirected to redirect URI, with the following parameters in the query string:

- `code` authorization code
- `state` state parameter sent in the original request. You should compare this value with the value stored in the user's session to ensure the authorization code obtained is in response to requests made by this client rather than another client application.

Step two

The client now sends a POST request to `/auth/user/access_token` or `/auth/contact/access_token` with the following parameters in request body:

- `grant_type` with the value of `authorization_code`
- `client_id` with the client identifier
- `scope` a space delimited list of scopes
- `client_secret` with the client secret

- `redirect_uri` with the same redirect URI the user was redirect back to
- `code` with the authorization code from the query string

Instead of specifying `client_id` and `client_secret` in request body, you can send HTTP Basic Authorization header, using the client ID as username and client secret as password.

The server will respond with a JSON object containing the following properties:

- `token_type` with the value `Bearer`
- `expires_in` with an integer representing the TTL of the access token
- `access_token` access token - a JWT signed with the server's private key
- `refresh_token` an encrypted payload that can be used to refresh the access token when it expires.

Access token can be now used to make calls to protected endpoints.

Implicit Grant

The implicit grant is similar to the authorization code grant with two distinct differences.

It is intended to be used for user-agent-based clients (e.g. single page web apps) that can't keep a client secret because all of the application code and storage is easily accessible.

Secondly instead of the authorization server returning an authorization code which is exchanged for an access token, the authorization server returns an access token.

Flow

The client will redirect the user's web browser to `/auth/user/authorize` or `/auth/contact/authorize` with the following parameters in the query string:

- `response_type` with the value `token`
- `client_id` with the client identifier
- `scope` a space delimited list of scopes
- `redirect_uri` with the client redirect URI. This parameter is optional, and defaults to first URI from the list configured by 1CRM administrator. If specified, this parameter must match one of redirect URIs configured by 1CRM administrator.
- `state` optional CSRF token. This parameter is optional but highly recommended. You should store the value of the CSRF token in the user's session to be validated when they return to your application

The user will be asked to login and approve the client. If the user approves the client, they will be redirected to redirect URI, with the following parameters in the query string:

- `token_type` with the value `Bearer`
- `expires_in` with an integer representing the TTL of the access token
- `access_token` access token - a JWT signed with the authorization server's private key
- `state` with the state parameter sent in the original request. You should compare this value with the value stored in the user's session to ensure the authorization code obtained is in response to requests made by this client rather than another client application.

Note: This grant does not return refresh token.

Resource Owner Password Credentials Grant

This grant provides great user experience for web applications and native mobile applications, because it does not require the user to be redirected to 1CRM for password entry. This grant is only available for private clients.

Flow

The client will send a a POST request to `/auth/user/access_token` or `/auth/contact/access_token` with the following parameters in request body:

- `grant_type` with the value of `password`
- `client_id` with the client identifier
- `scope` a space delimited list of scopes
- `client_secret` with the client secret
- `username`
- `password`

Instead of specifying `client_id` and `client_secret` in request body, you can send HTTP Basic Authorization header, using the client ID as username and client secret as password.

The server will respond with a JSON object containing the following properties:

- `token_type` with the value `Bearer`
- `expires_in` with an integer representing the TTL of the access token
- `access_token` access token - a JWT signed with the server's private key
- `refresh_token` an encrypted payload that can be used to refresh the access token when it expires.

Access token can be now used to make calls to protected endpoints.

Client Credentials Grant

To enable this grant, 1CRM administrator assigns a user to API client using 1CRM admin interface. After obtaining access token, API client can make requests to the API on behalf of that user. This grant is only available for Resource Owner of User type.

Flow

The client will send a a POST request to `/auth/user/access_token` or with the following parameters in request body:

- `grant_type` with the value of `client_credentials`
- `client_id` with the client identifier
- `scope` a space delimited list of scopes
- `client_secret` with the client secret

Instead of specifying `client_id` and `client_secret` in request body, you can send HTTP Basic Authorization header, using the client ID as username and client secret as password.

The server will respond with a JSON object containing the following properties:

- `token_type` with the value `Bearer`
- `expires_in` with an integer representing the TTL of the access token
- `access_token` access token - a JWT signed with the server's private key

Access token can be now used to make calls to protected endpoints.

Refresh Token Grant

Access tokens eventually expire; however some grants respond with a refresh token which enables the client to refresh the access token.

Flow

The client will send a a POST request to `/auth/user/access_token` or `/auth/contact/access_token` with the following parameters in request body:

- `grant_type` with the value of `refresh_token`
- `refresh_token` with the refresh token
- `client_id` with the client identifier
- `scope` a space delimited list of scopes
- `client_secret` with the client secret

Instead of specifying `client_id` and `client_secret` in request body, you can send HTTP Basic Authorization header, using the client ID as username and client secret as password.

The server will respond with a JSON object containing the following properties:

- `token_type` with the value `Bearer`
- `expires_in` with an integer representing the TTL of the access token
- `access_token` access token - a JWT signed with the server's private key
- `refresh_token` an encrypted payload that can be used to refresh the access token when it expires.

Session authentication

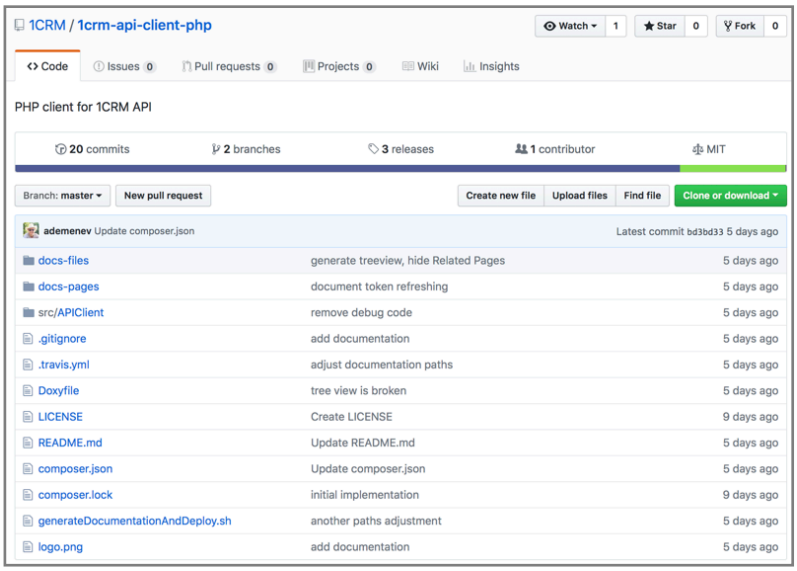
Session authentication is an authentication method for embedded 1CRM applications. Authentication is performed by adding the `Session ID` from browser cookies to HTTP requests when calling endpoints that require authentication. The request would be automatically authenticated using the credentials from the current 1CRM session with all user's ACL rights.

Note that Session Authentication is disabled by default.

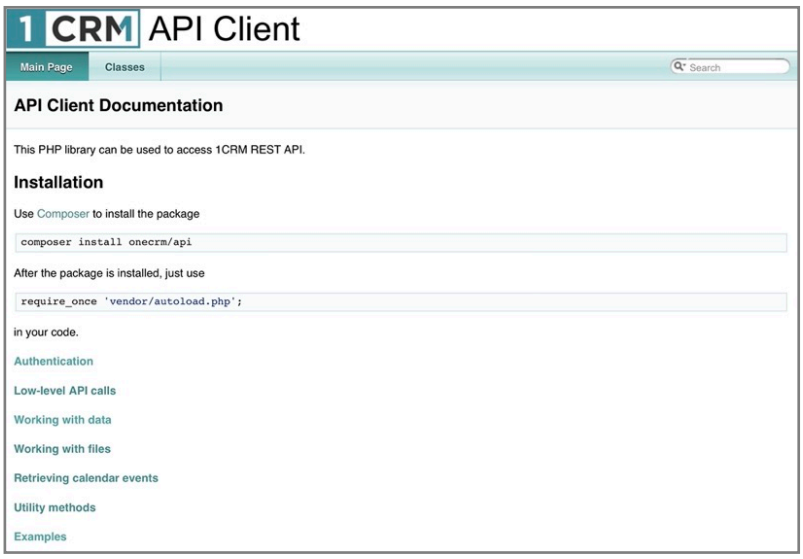
2.2.2 API Client Library for PHP

An API Client library, sometimes called a helper library, is a set of code that application developers can add to their development projects. It provides chunks of code that do the basic things an application needs to do in order to interact with an API - in this case the 1CRM REST API. Client libraries are provided to make a developer's life easier, to ensure applications follow best practices when using the API, and to make any code using the API more robust.

1CRM has created an open source API Client library for PHP which may be found here on github.



Documentation for the API Client library is here.



2.2.3 Types

Bool

This type represents a boolean value. Strings `yes`, `1` and `true` are recognized as `true`, and `no`, `0` and `false` are recognized as `false`. When sending parameters of this type in request body, prefer using JSON values of `true` or `false` instead of strings.

Int

This type represents an integer value. Parameters of this type can have limits set for minimum and maximum accepted values

String

This type represents a generic string value. Parameters of this type can have limits set for minimum and maximum accepted string length and/or regular expression that the string should match

Enum

This type represents a string that can only take one of predefined values

Float

This type represents a floating point numeric value. Parameters of this type can have limits set for minimum and maximum accepted values

Array

This type represents an array of values. Parameters of this type can be either a generic array without a predefined element type, or a typed array that should have only values of specific type as array elements

Object

This type represents a data structure known in different programming languages as associative array, map, symbol table, or dictionary. Object keys are always strings. Parameters of this type can have *schema* to specify accepted keys and value types for those keys

Date

Inherits String

This type represents a date value. The value must conform to `Y-m-d` format as used by PHP date function

DateTime

Inherits String

This type represents a date/time value. The value must conform to `Y-m-d H:i:s` format as used by PHP date function. Use GMT timezone.

Filename

Inherits String

This type represents a file name. The value should not contain any path information.

OneOf

This is a polymorphic type that can be one of several predefined types.

WebhookFilter

Inherits Object

This type represents a webhook filter.

Schema

Name	Type	Description	
glue	Enum {or, and}		!
conditions	Array	Depends	!
↳[n]	OneOf (StringFilter, DateFilter, DateTimeFilter, EnumFilter, NumberFilter, BoolFilter)		

StringFilter

Inherits Object

This type represents a webhook filter by a string field.

filter can be one of the following:

- eq field value is equal to value
- not_eq field value is not equal to value
- prefix field value starts with value
- suffix field value ends with value
- like field value contains value
- not_like field value does not contain value
- empty field value is empty
- not_empty field value is not empty

Schema

Name	Type	Description	
field	String	Name of the field this filter applies to	!
filter	Enum {eq, not_eq, prefix, suffix, like, not_like, empty, not_empty}	Filter operator	!
value	String	Operand	
when	WebhookWhen	Type of change	!

NumberFilter

Inherits Object

This type represents a webhook filter by a numeric field.

`filter` can be one of the following:

- `lt` field value is less than `value`
- `gt` field value is greater than `value`
- `lte` field value is less than or equal to `value`
- `gte` field value is grater than or equal to `value`
- `between` field value is between `value` and `value2`, inclusive
- `empty` field value is not set
- `not_empty` field value is set
- `eq` field value is equal to `value`
- `not_eq` field value is not equal to `value`

Schema

Name	Type	Description	
field	String	Name of the field this filter applies to	!
filter	Enum {lt, gt, lte, gte, between, empty, not_empty, eq, not_eq}	Filter operator	!
value	Float	Operand	
value2	Float	Second operand	
when	WebhookWhen	Type of change	!

DatetimeFilter

Inherits Object

This type represents a webhook filter by a date or datetime field.

`filter` can be one of the following:

- `not_empty` field value is not empty
- `empty` field value is empty
- `before_date` field value is before the date passed in `value`
- `after_date` field value is after the date passed in `value`
- `on_date` field value equals the date passed in `value`
- `not_on_date` field value does not equal the date passed in `value`
- `yesterday` field value is yesterday's date
- `today` field value is today's date
- `tomorrow` field value is tomorrow's date
- `between_dates` field value is between dates passed in `value` and `value2`, inclusive

You can also pass `period_prev`, `period_current`, or `period_next` in `filter`. In this case filed value will be checked against previous, current, or next period relative to current date. Period length is determined by `value`:

- `day` one day
- `week` calendar week
- `month` calendar month
- `quarter` calendar quarter
- `year` calendar year
- `fiscal_quarter` fiscal quarter
- `fiscal_year` fiscal year
- `days_N` N days

Schema

Name	Type	Description	
field	String	Name of the field this filter applies to	!
filter	Enum {not_empty, empty, before_date, after_date, on_date, not_on_date, yesterday, today, tomorrow, period_prev, period_current, period_next, between_dates}	Filter operator	!
value	OneOf (Date, DateTime)	Operand	
value2	OneOf (Date, DateTime)	Second operand	
when	WebhookWhen	Type of change	!

EnumFilter

Inherits Object

This type represents a webhook filter by a enum or multienum field.

`filter` can be one of the following:

- `eq` field value equals `value`; pass a string for enum field or an array of strings for multienum field
- `not_eq` field value not equals `value`; pass a string for enum field or an array of strings for multienum field
- `any_of` for enum fields, field value is contained in `value` array; for multienum fields, field value contains at least one element from array passed in `value`
- `not_any_of` for enum fields, field value is not contained in `value` array; for multienum fields, field value does not contain any of the elements from array passed in `value`
- `all_of` all elements passed in `value` array are contained in field value; applicable to multienum fields only
- `empty` field value is empty
- `not_empty` field value is not empty

Schema

Name	Type	Description	
field	String	Name of the field this filter applies to	!
filter	Enum {eq, not_eq, any_of, not_any_of, all_of, empty, not_empty}	Filter operator	!
value	OneOf (String, Array {String})	Operand. When filtering on enum fields, pass a string. For multienum fields, pass an array of strings	
when	WebhookWhen	Type of change	!

BoolFilter

Inherits Object

This type represents a webhook filter by a boolean field.

Schema			
Name	**Type**	**Description**	
field	String	Name of the field this filter applies to	!
filter	Enum {is_true, is_false}	Filter operator	!
when	WebhookWhen	Type of change	!

WebhookWhen

Inherits Enum {changed, not_changed, changed_to, current_value, prev_value}

This type represents the type of change webhook filter reacts to.

- `changed` Field value was changed. Actual previous and new values are not important
- `not_changed` Field value was not changed.
- `changed_to` Field was changed to a value that matches the criteria defined by filter's `operator`
- `current_value` Field's current value matches the criteria defined by filter's `operator`, regardless of whether it was changed or not.
- `prev_value` Field was changed, and its previous value matches the criteria defined by filter's `operator`

TestComplex

Inherits Object

This type exists solely for testing purposes. Do not use.

Schema		
Name	**Type**	**Description**
x	String	!
y	Int	
z	TestComplex	
q	DateTime	!

2.2.4 Authentication

Authorization request for contact

GET /auth/contact/authorize

See OAuth 2.0

Parameters			
Name	**Type**	**Description**	
response_type	Enum {code, token}	Expected response type	Q
client_id	String	API client identifier	Q
redirect_uri	String	This parameter is optional, if not specified, the user will be redirected to a pre-registered redirect URI	Q
scope	String	A space delimited list of scopes	Q
state	String	CSRF token. This parameter is optional but highly recommended. You should store the value of the CSRF token in the user's session to be validated when they return	Q

Authorization request for user

GET /auth/user/authorize

See OAuth 2.0

Parameters			
Name	**Type**	**Description**	
response_type	Enum {code, token}	Expected response type	Q
client_id	String	API client identifier	Q
redirect_uri	String	This parameter is optional, if not specified, the user will be redirected to a pre-registered redirect URI	Q
scope	String	A space delimited list of scopes	Q
state	String	CSRF token. This parameter is optional but highly recommended. You should store the value of the CSRF token in the user's session to be validated when they return	Q

Authorization grant for contact

`POST` `/auth/contact/access_token`

See OAuth 2.0

Parameters			
Name	**Type**	**Description**	
grant_type	Enum {client_credentials, password, authorization_code, refresh_token}	Grant type	B
client_id	String	API client identifier	B
client_secret	String	API client secret	B
refresh_token	String	Refresh token	B
redirect_uri	String	This parameter is optional, if not specified, the user will be redirected to a pre-registered redirect URI	B
code	String	Authorization code	B

Return			
Name	**Type**	**Description**	
token_type	String	Access token type. Always `Bearer`	!
expires_in	Int	An integer representing the TTL of the access token	!
access_token	String	A JWT signed with the authorization server's private key	!
refresh_token	String	An encrypted payload that can be used to refresh the access token when it expires	
state	String	State parameter sent in the original request. You should compare this value with the value stored in the user's session to ensure the authorization code obtained is in response to requests made by this client rather than another client application	

Authorization grant for user

POST `/auth/user/access_token`

See OAuth 2.0

Parameters			
Name	**Type**	**Description**	
grant_type	Enum {client_credentials, password, authorization_code, refresh_token}	Grant type	B
client_id	String	API client identifier	B
client_secret	String	API client secret	B
refresh_token	String	Refresh token	B
redirect_uri	String	This parameter is optional, if not specified, the user will be redirected to a pre-registered redirect URI	B
code	String	Authorization code	B

Return			
Name	**Type**	**Description**	
token_type	String	Access token type. Always `Bearer`	!
expires_in	Int	An integer representing the TTL of the access token	!
access_token	String	A JWT signed with the authorization server's private key	!
refresh_token	String	An encrypted payload that can be used to refresh the access token when it expires	
state	String	State parameter sent in the original request. You should compare this value with the value stored in the user's session to ensure the authorization code obtained is in response to requests made by this client rather than another client application	

2.2.5 Working with data

Add or remove a record from user's favorites

POST /data/favorites/{model}/{id}

Parameters			
Name	**Type**	**Description**	
model	String [1:]	Model	P !
id	String [1:]	Record ID	P !

Return		
Name	**Type**	**Description**
status	Int	New favorite status. 0 if added, 1 if removed

Get list of records

GET /data/{model}

Retrieve list of records belonging to specified `model`

Returned objects do not have a predefined structure, it depends on model and requested fields. See Metadata

Parameters

Name	Type	Description	
model	String [1:]	Model to query	P !
fields	Array	Array of field names to fetch. When omitted, a limited number of fields are returned, depending on model. Record ID is guaranteed to be returned, and for most models, `name` and/or `_display` fields.	Q
↳[n]	String		Q
query_favorite	Bool	Query favorites. If set, `favorite` field will be added to each result row, indicating whether the current user added the item to Favorites. Note that item is in favorites if and only if the value of `favorite` field is zero	Q
filter_text	String	Generic search string. Fields involved in search depend on model	Q
filters	Object {String}	Filters to apply. To get list of available filters for a model, use metadata	Q
order	String	Sort order	Q
offset	Int [0:] (0)	Offset in the list to start retrieval from	Q
limit	Int [1:200] (20)	Limits number of records returned	Q

Return

Name	Type	Description	
records	Array	Array of retrieved records	!
↳[n]	Object		
total_results	Int	Total number of results, without taking offset and limit in account	!

Create records

POST /data/{model}

Create a record of specified `model`

Parameters			
Name	**Type**	**Description**	
model	String [1:]	Model to query	P !
data	Object	An object with keys matching field names to update. Fields not defined in this object are not modified	B !

Return			
Name	**Type**	**Description**	
id	String	ID of created record	!

Get single record

GET /data/{model}/{id}

Retrieve single record belonging to specified `model`, identified by `id`

Returned object does not have a predefined structure, it depends on model and requested fields. See Metadata

Parameters

Name	Type	Description	
model	String [1:]	Model to query	P !
id	String [1:]	Record ID	P !
fields	Array	Array of field names to fetch. When omitted, all available fields are returned.	Q
↳[n]	String		

Return

Name	Type	Description	
record	Object	Retrieved record	!

Update a record

PATCH `/data/{model}/{id}`

Update record belonging to specified `model`, identified by `id`

Parameters			
Name	**Type**	**Description**	
model	String [1:]	Model to query	P !
id	String [1:]	Record ID	P !
data	Object	An object with keys matching field names to update. Fields not defined in this object are not modified	B !

Return			
Name	**Type**	**Description**	
result	Bool	Always true	!

Delete a record

DELETE `/data/{model}/{id}`

Delete record belonging to specified `model`, identified by `id`

Parameters			
Name	**Type**	**Description**	
model	String [1:]	Model	P !
id	String [1:]	Record ID	P !

Return			
Name	**Type**	**Description**	
result	Bool	True if deleted successfully	!

Get list of related records

GET /data/{model}/{id}/{link}

Retrieve list of related records belonging to specified `model` and `id` via specific `link`

Returned objects do not have a predefined structure, it depends on model and requested fields. See Metadata

Parameters

Name	Type	Description	
model	String [1:]	Model to query	P !
id	String [1:]	Record ID	P !
link	String [1:]	Link name	P !
fields	Array	Array of field names to fetch. When omitted, a limited number of fields are returned, depending on model. Record ID is guaranteed to be returned, and for most models, `name` and/or `_display` fields.	Q
↳[n]	String		
filter_text	String	Generic search string. Fields involved in search depend on model	Q
filters	Object {String}	Filters to apply. To get list of available filters for a model, use metadata	Q
order	String	Sort order	Q
offset	Int [0:] (0)	Offset in the list to start retrieval from	Q
limit	Int [1:200] (20)	Limits number of records returned	Q

Return

Name	Type	Description	
records	Array	Array of retrieved records	!
↳[n]	Object		
total_results	Int	Total number of results, without taking offset and limit in account	!

Add related records

POST /data/{model}/{id}/{link}

Add list of related records belonging to specified `model` and `id` via specific `link`.

You can specify related records in 2 ways:

1. Pass an array of related IDs in `records` parameter. This is suitable for most links, where you just want to set relationship between records, without specifying additional link data. Example: linking Contact to Account.
2. Pass an array of objects with additional data to be inserted into join table in `records_with_data` parameter. For example, when adding a product to assembly, you want to specify the quantity of products in the assembly. Imagine you want to add 5 products with ID `3d3e96d1-8d7c-acd6-e338-55b9b0cc5aae` to assembly with ID `5121670a-9ddb-8330-195d-5979fc9a6906`. Then you POST to `/data/Assembly/5121670a-9ddb-8330-195d-5979fc9a6906/products`, and pass the following JSON in request body:

```
{

   "records_with_data": [
       {
           "id": "3d3e96d1-8d7c-acd6-e338-55b9b0cc5aae",
           "quantity" : 5
       }
   ]
}
```

Parameters

Name	Type	Description	
model	String [1:]	Parent Model	P !
id	String [1:]	Parent ID	P !
link	String [1:]	Link name	P !
records	Array	Array of related record IDs to be added to specified link.	B
↳[n]	String		
records_with_data	Array	Array of objects, each representing additional data to be inserted into join table	B
↳[n]	Object		

Remove relationship between records

DELETE /data/{model}/{id}/{link}/{rel_id}

Parameters			
Name	**Type**	**Description**	
model	String [1:]	Parent Model	P !
id	String [1:]	Parent record ID	P !
link	String [1:]	Link name	P !
rel_id	String [1:]	Related record ID	P !

Return			
Name	**Type**	**Description**	
Result	Bool	True if deleted successfully	!

Get list of all related records

GET /data/link/{model}/linked-records/{link}

Retrieve list of related records belonging to specified `model` via specific `link`

Returned objects do not have a predefined structure, it depends on model and requested fields. See Metadata

Parameters			
Name	**Type**	**Description**	
model	String [1:]	Model to query	P !
link	String [1:]	Link name	P !
fields	Array	Array of field names to fetch. When omitted, a limited number of fields are returned, depending on model. Record ID is guaranteed to be returned, and for most models, `name` and/or `_display` fields.	Q
↳[n]	String		

Parameters			
order	String	Sort order	Q
offset	Int [0:] (0)	Offset in the list to start retrieval from	Q
limit	Int [1:200] (20)	Limits number of records returned	Q

Return			
Name	**Type**	**Description**	
records	Array	Array of retrieved records	!
↳[n]	Object		
total_results	Int	Total number of results, without taking offset and limit in account	!

Erase Personal Data

PATCH `/data/erase_personal/{model}/{id}`

Erase Personal Data for specified model identified by `id`

Parameters			
Name	**Type**	**Description**	
model	String [1:]	Model to query	P !
id	String [1:]	Record ID	P !
data	Object	An object with `fields` property with the comma separated fields string for erase. Fields not added to this string are not erased.	B !

Return			
Name	**Type**	**Description**	
Result	Bool	Always true	!

2.2.6 Tally Objects

1CRM includes a set of modules that have a common property: each record from those modules contains a list of related **line items**. This includes records that represent financial transactions (Quotes, Invoices, Bills etc.) and the moving of goods (Shipping, Receiving). Records from such modules are collectively referred to as **Tally Objects**. 1CRM API provides a set of endpoints for working with such objects that allow developers to treat the Tally objects as a whole, not as separate *parent* record (for example, invoice) and a set of line items.

Get a Tally Object

GET `/tally/{model}/{id}`

Parameters			
Name	**Type**	**Description**	
model	String [1:]	Model to query	P !
id	String [1:]	Record ID	P !

Return		
Name	**Type**	**Description**
record	Object	*Parent* record as if it was retrieved using `/data/{model}/{id}` endpoint
groups	Array	Array of groups, each group contains one or more line items

2.2.7 Working with calendars

Get list of events

GET /calendar/events

Retrieve list of events within specified dates range. Returned records are grouped by type, and within each type records are sorted by date. No more than 200 records of each type are returned.

Parameters

Name	Type	Description	
start_date	DateTime	Lower date bound	Q !
end_date	DateTime	Upper date bound	Q !
types	Array	When present, this parameter limits returned events to specified types	Q
↳[n]	Enum {Call, Meeting, Task, ProjectTask}		

Return

Name	Type	Description	
records	Array	Array of retrieved records	!
↳[n]	Object		
↳[id]	String	Record ID	!
↳[start_date]	DateTime	Event start date/time	
↳[due_date]	DateTime	Event due date/time	
↳[name]	String	Event name	!
↳[type]	Enum {Call, Meeting, Task, ProjectTask}	Event type	!
↳[location]	String	Event location, if applicable	

2.2.8 Working with metadata

Get fields definition for a model

GET /meta/fields/{model}

Parameters			
Name	**Type**	**Description**	
model	String [1:]	Model to query	P !

Return			
Name	**Type**	**Description**	
fields	Array	An array with fields definitions	!
filters	Array	An array with filters definitions	

Get locale-related information

GET /meta/locale

Retrieve information about how 1CRM is configured to format numbers, addresses and currency values.

Date and time formats are as defined in the documentation of PHP date() function.

Name format can contain *s*, *f* and *l* placeholders, for *salutation*, *first name* and *last name*, respectively.

Address format can contain placeholders. A placeholder consists of a letter between curly braces:

- *{s}* for street address
- *{c}* for city
- *{t}* for state
- *{o}* for country
- *{p}* for postal code
- *{a}* for company name
- *{n}* for contact name
- *{d}* for department
- *{h}* for phone
- *{f}* for fax

| (pipe symbol) denotes a space, [ ] (a space in square bracket) is for new line.

Return

Name	Type	Description	
date_format	String	Date format	!
time_format	String	Time format	!
address_format	Object	Address format	!
↳[display]	String	Address format descriptive name	!
↳[format]	String	Address format	!
number_format	Object	Number format	!
↳[display]	String	Number format descriptive name	!
↳[dec_sep]	String	Deciamal part separator	!
↳[grp_sep]	String	Groups separator	!
base_currency	Object	Base currency	!
↳[name]	String	Currency name	!
↳[iso4217]	String	Currency code according to ISO 4217	!
↳[symbol]	String	Currency symbol	!
↳[significant_digits]	Int	Number of symbols after decimal point	!
↳[symbol_place_after]	Bool	`true` if symbol must be placed after amount in monetary values, `false` if the symbol goes before amount	!
↳[space_separate]	Bool	`true` if a space is required between numeric value and currency symbol	!

Get list of available modules

GET /meta/modules

Get list of available modules (models).

Return			
Name	**Type**	**Description**	
list	Array	List of modules	!
↳[n]	Object		
↳[module]	String	Module name	!
↳[primary_model]	String	Module's primary model	!
↳[models]	Array	Models	!
↳[n]	String	Model name	

Get ListView metadata for a model

GET /meta/list_view/{model}

Retrieve ListView metadata belonging to specified model

Parameters			
Name	**Type**	**Description**	
model	String [1:]	Model to query	P !
layout	String	Get metadata for this layout	B

Return			
Name	**Type**	**Description**	
current_layout	Array	An array with current ListView layout (tab) params	!
layouts	Array	Array of all ListView layouts for the module	!
↳[n]	Object		
settings	Object	An array with Listview layout settings	!
hooks	Array	An array with ListView hooks	
columns	Array	Array of ListView columns	!
↳[n]	Object		
filters	Array	Array of filters for the current layout	!
↳[n]	Object		
hidden fields	Array	Array of hidden fields	
↳[n]	Object		
mass_update	Array	Mass Update details - buttons and fields	
↳[n]	Object		
↳[buttons]	Array	ListView mass update buttons (actions)	
↳[fields]	Array	Mass update form fields	

Get Sub Panels metadata for a model

GET /meta/sub_panels/{model}/{detail_layout}

Retrieve Sub Panels metadata belonging to specified model and detail_layout

Parameters			
Name	**Type**	**Description**	
model	String [1:]	Model to query	P !
detail_layout	String [1:]	DetailView layout tab	P !

Return			
Name	**Type**	**Description**	
panels	Array	A list of SubPanels with metadata	!
↳[n]	Object		
↳[name]	String	SubPanel name	!
↳[module]	String	SubPanel module	!
↳[bean_name]	String	SubPanel model name	!
↳[model_type]	String	SubPanel model type	!
↳[managed]	Bool		
↳[removeable]	Bool		
↳[updateable]	Bool		
↳[no_create]	Bool	Hide Create button if true	
↳[icon]	String	Custom icon	
↳[detail_link_url]	String	Custom DetailView URL	
↳[detail_link_action]	String	Custom DetailView Action	
↳[layout]	String	SubPanel layout name	
↳[layout_type]	String	SubPanel layout type	
↳[title]	String	Title lang constant	

Return		
↳[target_key]	Bool	Target Key
↳[source_key]	Bool	Source Key
↳[custom_buttons]	Array	An array with SubPanel custom buttons
↳[columns]	Array	An array with SubPanel columns

Get DetailView metadata for a model

GET /meta/detail_view/{model}/{layout}

Retrieve DetailView metadata belonging to specified model and layout

Parameters			
Name	**Type**	**Description**	
model	String [1:]	Model to query	P !
layout	String [1:]	Selected DetailView layout	P !

Return			
Name	**Type**	**Description**	
title	String	Form title	!
current_layout	Array	An array with current DetailView layout (tab) params	!
layouts	Array	An Array of all DetailView layouts(tabs) for the module	!
↳[n]	Object		
buttons	Array	An Array of custom module DetailView buttons	
↳[n]	Object		
scripts	Array	An array of custom scripts which should be included	
form_hooks	Array	An array of custom form hooks	
↳[n]	Object		
summary	Object	Summary part of the detail page	
sections	Array	An array of detail page sections	!
↳[n]	Object		

Get generic module metadata

GET /meta/generic_module_layouts/{module}

Generic module metadata - default list, detail, popup layouts

Parameters			
Name	**Type**	**Description**	
module	String [1:]	Module	P !

Return			
Name	**Type**	**Description**	
list	Array	Default ListView layout and view	!
popup	Array	Default Popup layout and view	!
detail	Array	Default DetailView layout and view	!

2.2.9 Working with files

Upload a small file

POST /files/upload/base64

Upload a file. Uploaded file will be saved to a temporary location. Returned file ID can be used as a value for fields having `file_ref` or `image` type when creating or updating records.

This endpoint is suitable for uploading relatively small files, such as contact or user photos. Request body length should not exceed 1048576 bytes.

Note that uploaded files will be kept on server for a limited amount of time if not linked in `file_ref` or `image` field after upload.

Parameters

Name	Type	Description	
model	String [1:]	File name. Should not contain any path information	B !
mimetype	String [1:]	File MIME type	B !
data	String	File data, `base64` encoded	B !

Return

Name	Type	Description	
id	String	Uploaded file ID	!

Upload a file

POST /files/upload

Upload a file. Uploaded file will be saved to a temporary location. Returned file ID can be used as a value for fields having `file_ref` or `image` type when creating or updating records.

This endpoint is suitable for uploading larger files, compared to `/files/upload/base64`. File size limit depends on maximum post size defined in PHP configuration.

Note that uploaded files will be kept on server for a limited amount of time if not linked in `file_ref` or `image` field after upload.

Parameters

Name	Type	Description	
CONTENT_TYPE	String (application/octet-stream)	File content type	H
CONTENT_LENGTH	Int [0:]	File size	H !
X_ONECRM_FILENAME	Filename [1:]	File name	H !

Return

Name	Type	Description	
id	String	Uploaded file ID	!

Download a file

GET /files/download/{model}/{id}

Download a file.

This endpoint is for downloading 1CRM Documents, Note attachments and temporary uploaded files (see /files/upload/base64).

File source is identified with `model` and `id` parameters:

- When `model` equals to `Document`, latest document revision will be downloaded. `id` is the Document ID
- When `model` equals to `DocumentRevision`, specific document revision will be downloaded. `id` is the Document Revision ID
- When `model` equals to `Notes`, note attachment will be downloaded, `id` is Note ID
- When `model` equals to `upload`, contents of temporary uploaded file will be downloaded. `id` is the ID returned from upload endpoint

On success, the response body contains raw file data. Additional information may be returned in `Content-Type`, `Content-Length`, `Content-Disposition`, and `X-OneCRM-Filename` response headers.

On failure, HTTP response code different from 200 will be returned, and response body contains additional information in JSON format.

Parameters

Name	Type	Description	
model	Enum	Specifies model	P !
id	String [1:]	Specifies ID	P !

Get information about a file

GET `/files/info/{model}/{id}`

Get information about a file.

This endpoint is for Retrieving metadata for 1CRM Documents, Note attachments and temporary uploaded files (see /files/upload/base64).

See /files/download/:model/:id for description of `model` and `id` parameters.

Parameters

Name	Type	Description	
model	Enum	Specifies model	P !
id	String [1:]	Specifies ID	P !

Return

Name	Type	Description	
name	String	File name	!
mimetype	String	File MIME type	!
modified	Int	File modification time, in secons since UNIX epoch	!
size	Int	File size in bytes	!
temp_url	String	Temporary download URL	

2.2.10 Webhooks

A Webhook is an HTTP callback: an HTTP POST that occurs when something happens; a simple event-notification via HTTP POST.

In 1CRM, 3 types of webhooks are available create, update and create_update. Webhook type determines the event that triggers the notification. create webhooks are triggered when a new record is created. update webhooks are triggered when an existing record is updated. create_update webhooks are triggered for both new records and updated records.

When a webhook is triggered, a HTTP POST request is sent to specified URL. Content-Type header is set to application/json, and a JSON object is passed in request body. The object contains the values of record's fields after the update.

When a record is updated, first all webhooks with corresponding type and model are selected. Then each webhook is examined for the user that created it. If the user has no read access to the updated record, the webhook will be excluded from further examination. Next, for each webhook, each filter from filters is examined:

- if glue is and, and **all** conditions from filters are satisfied, the webhook triggers
- if glue is or, and **at least one** condition from filters is satisfied, the webhook triggers

Here are some examples of webhook filters construction. For these examples, the `Contact` model is used.

Basic filter structure

Webhook filter consists of 2 elements: `conditions` and `glue`. Conditions are an array of conditions to check, and `glue` tells how the conditions are to be combined.

`glue` can be either `and` or `or`. If `glue` is `and`, and **all** conditions are satisfied, the webhook triggers. If `glue` is `or`, and **at least one** condition is satisfied, the webhook triggers.

Basic filter structure:

```
{
    glue: "or", // "or", "and"
    conditions: [
        {
            field: "first_name",
            filter: "eq",
            value: "John",
            when: "current_value"
        }
    ]
}
```

This simple filter triggers the webhook if after update `first_name` equals to "John". Note that in this case `glue` can be either `or` or `and` - it does not matter when there is only one condition.

Combining multiple conditions

In previous example, the webhook would trigger if contact's first name is "John". Now imagine you want to trigger a webhook for every update to a contact whose first name is John, but only if their last name was changed to Smith. In this case the filter will look like this:

```
{
    glue: "and",
    conditions: [
        {
            field: "first_name",
            filter: "eq",
            value: "John",
            when: "current_value"
        },
        {
            field: "last_name",
            filter: "eq",
            value: "Smith",
            when: "changed_to"
        }
    ]
}
```

You can add as many conditions as you need:

```
{
    glue: "or",
    conditions: [
        {
            field: "last_name",
            filter: "eq",
            value: "Smith",
            when: "changed_to"
        },
        {
            field: "last_name",
            filter: "eq",
            value: "Johnson",
            when: "changed_to"
        },
        {
            field: "last_name",
            filter: "eq",
            value: "Black",
            when: "changed_to"
        }
    ]
}
```

Choosing values to check

As you must have noticed, we used different values for `when` in the conditions. This field tells which value you want to check: the old value (before the update) or the new value (after the update). Let's see how to use different `when` values.

```
{
    glue: "or",
    conditions: [
        {
            field: "last_name",
            filter: "eq",
            value: "Smith",
            when: "changed_to"
        },
    ]
}
```

With this filter, the webhook will trigger when last name changed to “Smith".

```
{
    glue: "or",
    conditions: [
        {
            field: "last_name",
            filter: "ne",
            value: "Smith",
            when: "changed_to"
        },
    ]
}
```

With this filter, the webhook will trigger when last name changed to anything other than "Smith". Note - if the name was not equal to "Smith" and it was not changed - the condition is not satisfied!

```
{
    glue: "or",
    conditions: [
        {
            field: "last_name",
            filter: "ne",
            value: "Smith",
            when: "current_value"
        },
    ]
}
```

With this filter, the webhook will trigger when last name after the update is not equal to "Smith". It does not matter if the last name was updated or not - the value after the update is tested.

```
{
    glue: "or",
    conditions: [
        {
            field: "last_name",
            filter: "eq",
            value: "Smith",
            when: "previous_value"
        },
    ]
}
```

With this filter, the webhook will trigger when last name was changed, and after the update it is equal to “Smith".

```
{
    glue: "or",
    conditions: [
        {
            field: "last_name",
            when: "changed"
        },
    ]
}
```

With this filter, the webhook will trigger when last name was changed. The actual new value does not matter, and the condition does not need `filter` and `value`.

```
{
    glue: "or",
    conditions: [
        {
            field: "last_name",
            when: "not_changed"
        },
    ]
}
```

With this filter, the webhook will trigger when last name was not changed. This condition does not need `filter` and `value`.

Operators

All conditions except `changed` and `not_changed` need an operator, passed in `filter`. Applicable operators depend on field type.

See NumberFilter, DatetimeFilter, EnumFilter, BoolFilterr, StringFilter for operators available for different field types.

Nested filters

As you can see, pretty sophisticated filters can be constructed. But there are cases when simply combining multiple conditions with `and` or `or` is not enough. In such cases, nested filters can be used. Let's continue with our Contact examples. Imagine you need a webhook triggered after contact's name is changed to either "John Smith" or "Jack Brown". In this case you construct your filter as follows:

```
{
    glue: "and",
    conditions: [
        {
            glue: "or",
            conditions: [
                {
                    field: "last_name",
                    when: "changed"
                },
                {
                    field: "first_name",
                    when: "changed"
                }
            ]
        },
        {
            glue: "or",
            conditions: [
                {
                    glue: "and",
                    conditions: [
                        {
                            field: "last_name",
                            when: "current_value",
                            filter: "eq",
                            value: "Smith"
                        },
                        {
                            field: "first_name",
                            when: "current_value",
                            filter: "eq",
                            value: "John"
                        }
                    ]
                },
                {
                    glue: "and",
                    conditions: [
                        {
                            field: "last_name",
                            when: "current_value",
                            filter: "eq",
                            value: "Brown"
                        },
                        {
                            field: "first_name",
                            when: "current_value",
                            filter: "eq",
                            value: "Jack"
                        }
                    ]
                }
            ]
        },
    ]
}
```

Get List of webhooks

GET /webhooks

Get List of webhooks.

Parameters			
Name	**Type**	**Description**	
offset	Int [0:] (0)	Offset in the list to start retrieval from	Q
limit	Int [1:200] (20)	Limits number of records returned	Q

Return			
Name	**Type**	**Description**	
total_results	Int	Total number of results, without taking offset and limit in account	!
records	Array	Webhook ID	
↳[n]	Object		
↳[id]	String	Webhook ID	!
↳[type]	String	Webhook type	!
↳[url]	String	Webhook URL	!
↳[model]	String	Model that triggers this webhook	!
↳[filters]	WebhookFilter	Model-specific filters	
↳[assigned_user_id]	String	User assigned to the webhook	!

Create a webhook

POST /webhooks

Parameters			
Name	**Type**	**Description**	
type	Enum {create, update, create_update}	Webhook type	B !
url	String [1:]	Webhook URL	B !
model	String [1:]	Model that triggers this webhook	B !
filters	WebhookFilter	Describes filters to trigger webhook on specific events only	B

Return			
Name	**Type**	**Description**	
id	String	Webhook ID	!
type	String	Webhook type	!
url	String	Webhook URL	!
model	String	Model that triggers this webhook	!
filters	WebhookFilter	Model-specific filters	

Delete a webhook

DELETE /webhooks/{id}

Delete webhook identified by `id`.

Parameters			
Name	**Type**	**Description**	
id	String [1:]	Webhook ID	P !

Return			
Name	**Type**	**Description**	
id	String	Webhook ID	!
type	String	Webhook type	!
url	String	Webhook URL	!
model	String	Model that triggers this webhook	!

Get a webhook

GET /webhooks/{id}

Get data for webhook identified by `id`.

Parameters			
Name	**Type**	**Description**	
id	String [1:]	Webhook ID	P !

Return			
Name	**Type**	**Description**	
id	String	Webhook ID	!
type	String	Webhook type	!
url	String	Webhook URL	!
model	String	Model that triggers this webhook	!
filters	WebhookFilter	Model-specific filters	

Update a webhook

PUT /webhooks/{id}

Parameters			
Name	**Type**	**Description**	
id	String [1:]	Webhook ID	P !
type	Enum {create, update, create_update}	Webhook type	B !
url	String [1:]	Webhook URL	B !
model	String [1:]	Model that triggers this webhook	B !
filters	WebhookFilter	Describes filters to trigger webhook on specific events only	B

Return			
Name	**Type**	**Description**	
id	String	Webhook ID	!
type	String	Webhook type	!
url	String	Webhook URL	!
model	String	Model that triggers this webhook	!
filters	WebhookFilter	Model-specific filters	

2.2.11 Portal

Create access token for portal user

POST /portal/auth

! This endpoint is not available when Basic authentication is used.

! Access to this endpoint is enabled for admin users only.

Create a new access token to be used for portal access. Returned access token allows access to a limited subset of data that is relevant for certain contact only.

Parameters

Name	Type	Description	
contact_id	String [1:]	Contact ID	B !

2.2.12 Utility

Get information about authenticated user

GET /me

Returns 1CRM version. Can be used to validate login info

Return			
Name	**Type**	**Description**	
version	String	1CRM version	!
products	Array	List if licensed products	!
↳[n]	String		
authenticated	Bool	True if the request contains valid authentication header	!

Get server public key

GET /public_key

Returns 1CRM version. Can be used to validate login info

Return			
Name	**Type**	**Description**	
key	String	Contents of server public key	!

Get 1CRM version

GET `/version`

Returns 1CRM version. Can be used to validate login info

Return			
Name	**Type**	**Description**	
version	String	1CRM version	!
products	Array	List if licensed products	!
↳[n]	String		
authenticated	Bool	True if the request contains valid authentication header	!

2.2.13 Print PDF

Print PDF

GET `/printer/pdf/{model}/{id}`

Parameters			
Name	**Type**	**Description**	
Model	String [1:]	Record model	P !
id	String [1:]	Record ID	P !

Print Personal Data PDF

GET `/printer/pdf/personal/{model}/{id}`

Parameters			
Name	**Type**	**Description**	
Model	String [1:]	Record model	P !
id	String [1:]	Record ID	P !

2.2.14 Audit

Get list of audit logs

GET `/audit/{model}`

Retrieve list of audit logs belonging to specified `model`

Parameters			
Name	**Type**	**Description**	
model	String [1:]	Model to query	P !
offset	Int [0:] (0)	Offset in the list to start retrieval from	Q
limit	Int [1:200] (20)	Limits number of records returned	Q

Return			
Name	**Type**	**Description**	
records	Array	Array of retrieved records	!
↳[n]	Object		
total_results	Int	Total number of results, without taking offset and limit	!

Get parent record audit logs

GET `/audit/{model}/{id}`

Retrieve list of audit logs belonging to specified `model`, identified by parent record `id`

Parameters

Name	Type	Description	
model	String [1:]	Model to query	P !
id	String [1:]	Parent record ID (audited record ID)	P !
offset	Int [0:] (0)	Offset in the list to start retrieval from	Q
limit	Int [1:200] (20)	Limits number of records returned	Q

Return

Name	Type	Description	
records	Array	Array of retrieved records	!
↳[n]	Object		
total_results	Int	Total number of results, without taking offset and limit	!

2.2.15 Reports

Get list of reports

GET `/reports/{model}`

Retrieve list of reports belonging to specified `model`

Parameters			
Name	**Type**	**Description**	
model	String [1:]	Model to query	P !
offset	Int [0:] (0)	Offset in the list to start retrieval from	Q
limit	Int [1:200] (20)	Limits number of records returned	Q

Return			
Name	**Type**	**Description**	
records	Array	Array of retrieved records	!
↳[n]	Object		
total_results	Int	Total number of results, without taking offset and limit	!

Get list of archived runs

GET `/reports/{model}/{id}`

Retrieve list of archived runs belonging to specified `model`, identified by report `id`

Parameters			
Name	**Type**	**Description**	
model	String [1:]	Model to query	P !
id	String [1:]	Report ID	P !
offset	Int [0:] (0)	Offset in the list to start retrieval from	Q
limit	Int [1:200] (20)	Limits number of records returned	Q

Return			
Name	**Type**	**Description**	
records	Array	Array of retrieved records	!
↳[n]	Object		
total_results	Int	Total number of results, without taking offset and limit	!

Get report data

GET /reports/data/{model}/{id}

Retrieve archived runs report data belonging to specified `model`, identified by run `id`

Parameters

Name	Type	Description	
model	String [1:]	Model to query	P !
id	String [1:]	Run ID	P !
offset	Int [0:] (0)	Offset in the list to start retrieval from	Q
limit	Int [1:200] (20)	Limits number of records returned	Q

Return

Name	Type	Description	
records	Array	Array of retrieved records	!
↳[n]	Object		
total_results	Int	Total number of results, without taking offset and limit	!

2.2.16 Working with list and detail layouts

Set ListView layout(tab) params for a model

POST /layouts/list_view/{model}

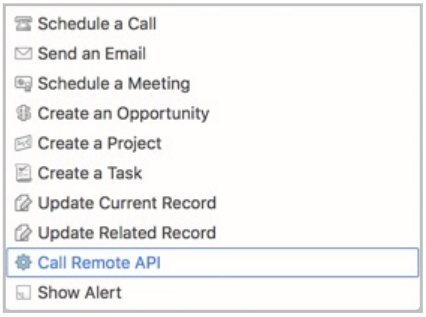
Set ListView layout(tab) params belong to specified `model`

Parameters			
Name	**Type**	**Description**	
model	String [1:]	Model	P !
layout	String [1:]	New selected ListView layout(tab)	B !
filter_layout	String [1:]	Filters form layout	B
filter_values	Array	Array of list filters values	B
offset	Int [0:] (0)	ListView offset value	B
limit	Int [1:200] (20)	ListView limit - number of records returned	B
order_by	String	Sort order	B

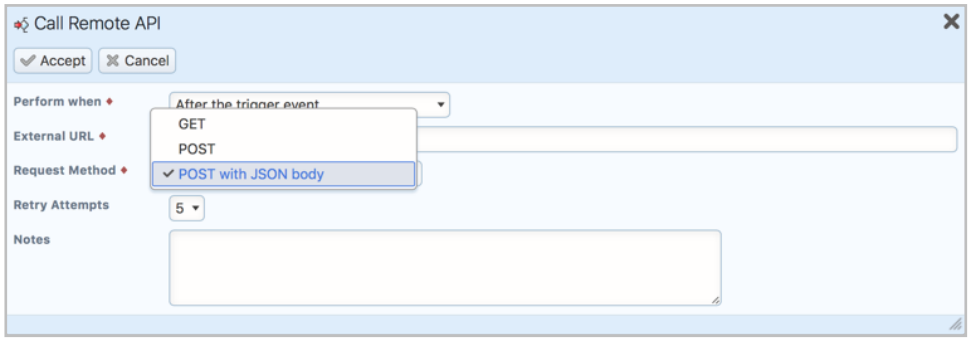
Return		
Name	**Type**	**Description**
return		

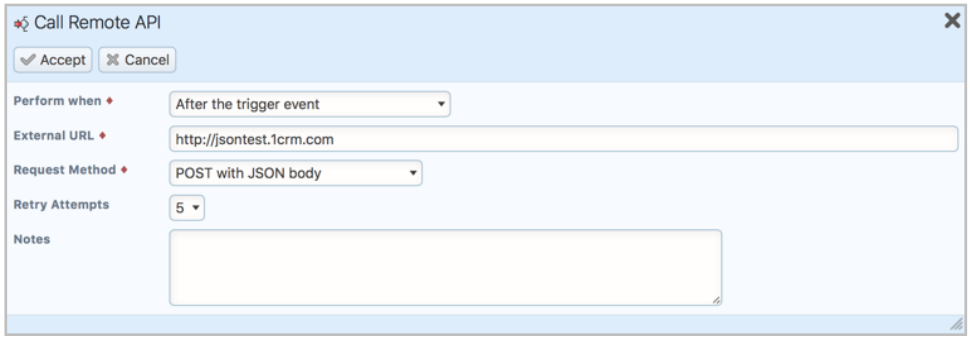
2.3 Integration using Workflow Actions

You can see an overview of how the Workflow capability in 1CRM works, in the Implementation Guide section 7.5.5. Fundamentally it allows 1CRM to automatically perform certain actions when various conditions are met. The range of actions includes:



As seen below, the *Call Remote API* action supports options for *GET*, *POST*, and *POST with JSON body*. When the action is performed 1CRM will link out to whatever URL is specified.





You will find a *POST* is useful for linking out to Zapier - the formdata type payload is accepted quite easily in Zapier. The *POST with JSON body* option is useful for linking to Microsoft Flow, for example.

When this workflow action is executed, it sends a request to the external URL specified. The request always contains the following fields:

module - the module that triggered the workflow
record - ID of the record that was modified
user - ID of user who triggered the workflow
source - always *workflow*

action - *saved* or *deleted*
when - *after* or *scheduled* (see below for details)
workflow - ID of workflow
operation - ID of workflow operation
tracker - see below for details

If the *when* field contains *after*, it means that the action is executed immediately after the workflow was triggered. If this field contains *scheduled*, it means that the action was configured to be executed with a delay after the workflow triggers. Note that this type of workflow action is always executed by a scheduler task, so *immediately* here means “the next time the scheduler runs after the workflow is triggered”.

The *tracker* field can be used by the remote server to check for duplicated actions. If a request to the remote API fails, 1CRM will retry the request later. There may be rare cases when the remote API call gets executed correctly, but because of a network error 1CRM does not receive a reply from the remote API. In this case 1CRM will resend the same request later. The remote API can use the *tracker* field to detect and ignore such duplicate requests.

The format of the HTTP request sent to the remote API depends on the request method configured. The *GET* method sends data in the query part of URL. The *POST* method will send the data in the request body using **`application/x-www-form-urlencoded`** content type. The method *POST with JSON body* sends the data in the request body using **`application/json`** content type.

In our example, 1CRM has the following values to send to the remote API, with the workflow action set to send HTTP requests to http://www.your-remote-api.com/action :

```
"module"    :  "Leads"
"record"    :  "4158ca78-6f11-ac08-3620-5aa7989bfe31"
"user"      :  "1"
"source"    :  "workflow"
"action"    :  "saved"
"when"      :  "scheduled"
"workflow"  :  "532457f5-5ed7-b15d-c4b9-5ab471a2cb0d"
"operation" :  "4d27b9f7-7a63-bdd3-f545-5ab6fa40b244"
"tracker"   :  "b7be4be0-5c5c-4af9-6768-5ab6fa492d36"
```

When using the *GET* method, 1CRM will send the following request:

```
GET /action?module=Leads&record=4158ca78-6f11-
ac08-3620-5aa7989bfe31&user=1&source=workflow&action=saved&when=scheduled&workflo
w=532457f5-5ed7-b15d-c4b9-5ab471a2cb0d&operation=4d27b9f7-7a63-bdd3-
f545-5ab6fa40b244&tracker=b7be4be0-5c5c-4af9-6768-5ab6fa492d36 HTTP/1.1
Host: www.your-remote-api.com
Accept: */*
```

When using the *POST* method, 1CRM will send the following request:

```
POST /action HTTP/1.1
Host: www.your-remote-api.com
Accept: */*
Content-Length: 245
Content-Type: application/x-www-form-urlencoded
```

```
module=Leads&record=4158ca78-6f11-
ac08-3620-5aa7989bfe31&user=1&source=workflow&action=saved&when=scheduled&workflo
w=532457f5-5ed7-b15d-c4b9-5ab471a2cb0d&operation=4d27b9f7-7a63-bdd3-
f545-5ab6fa40b244&tracker=b7be4be0-5c5c-4af9-6768-5ab6fa492d36
```

When using the *POST with JSON body* method, 1CRM will send the following request:

```
POST /action HTTP/1.1
Host: www.your-remote-api.com
Accept: */*
Content-Type: application/json
Content-Length: 283

{"module":"Leads","record":"4158ca78-6f11-
ac08-3620-5aa7989bfe31","user":"1","source":"workflow","action":"saved","when":"s
cheduled","workflow":"532457f5-5ed7-b15d-
c4b9-5ab471a2cb0d","operation":"4d27b9f7-7a63-bdd3-
f545-5ab6fa40b244","tracker":"b7be4be0-5c5c-4af9-6768-5ab6fa492d36"}
```

3.0 SugarCRM 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 1CRM. From the format of the PHP manifest file to the methods used for defining table models and views, the module architecture of 1CRM 7.0 and later releases is entirely different from that of SugarCRM.

Packages will need to be updated for compatibility by a competent PHP software developer, and in some cases entirely rewritten. 1CRM Systems Corp, via one of our 1CRM Partner organizations, can assist you with this sort of development work if required.

4.0 1CRM Module Development

4.1 Introduction

The process for developing custom modules for 1CRM 7+ is very different compared to previous versions, due to a set of fundamental changes in the 1CRM 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.

4.2 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 character, 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 from viewing by end-users. Comments are preceded by a hash (#) character.

```
<?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 subsequent 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/`.

4.3 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 the class. See Model Descriptors for details on these files, in particular the `bean_file` attribute on bean models.

Unlike in previous 1CRM versions, these SugarBean classes are not required for most database operations. Although the `retrieve()`/`save()` pattern may still be used, the preferred method 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 (including numbers, dates, and time values).

Each module directory will also contain several subdirectories:

Dashlets	This optional directory stores any custom Dashlets (widgets employed by the Home module) relating to the module.
`display`	This directory contains any Model Display Descriptor files. See section 4.6 for more information.
ext	For code extending existing modules. See section 5.1 for more information.
`language`	The location for any supporting language files for this module. See section 4.5 for more information.
`metadata`	Currently, the only file required in this directory is `module_info.php`, described below.
`models`	This directory contains any Model Descriptor files. See section 4.4 for more information.
`views`	This is the location for any Layout Descriptor files. See section 4.7 for more information.
widgets	Custom display widgets may be stored in this optional directory. See section 4.8 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.

For a custom module it may also be desirable to set the `icon` property. If you do, this should contain the path to a custom 16x16 icon file for the module. (A good practice can be to place this file at `Modules/NAME/ext/themes/Default/images/NAME.gif`.) This path needs to be relative to the install path, and after adding this property the 1CRM function *Admin — Maintenance — Rebuild Javascript Languages* needs to be performed to refresh the cached navigation data.

4.4 Model Descriptors

Previously represented by `vardefs.php`, in 1CRM 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 (under Administration > Maintenance) 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:

`type`	The descriptor type, also used in the filename prefix.
`bean_file`	In `bean` descriptors, the path to the SugarBean-derived class file represented by this model.
`primary_key`	The column or columns used to create the table's primary key.
`table_name`	The (unique) name of the table as represented in the SQL database.
`default_order_by`	The default column and order (ASC/DESC) used in sorting ListView results.

`display_name`	The column used to represent the displayed name of this record, for example when pointed to by a `ref` field or shown on the recently-viewed menu. Note that this property must be defined in order to properly display the title on a DetailView form. The display_name property may also be define a combination of fields, for example the display name of a Case consists of the case name with the case number as a prefix: display_name type: prefixed fields - case_number - name

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

`activity_log_enabled`	Setting this value to true causes changes to records in this module to be shown in the system activity log (dashlet).
`audit_enabled`	Enables auditing of database updates. Updates to fields also marked `audited` 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. The audit table is automatically created during a database repair operation.
`comment`	A text comment describing the function of the table.
`duplicate_merge`	To enable duplicate merging when a new record request appears similar to an existing record.
`optimistic_locking`	To enable optimistic locking for updates to this module. This feature is meant to warn users when others are performing updates to the same record simultaneously.
`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 with `unified_search: true` will be used to automatically filter relevant results.

4.4.1 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-derived 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). The `file` attribute should also be provided for classes which are not associated with a 1CRM model, and thus can't be included automatically.

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:

`new_record (`
`  RowUpdate &$update,`
`  array $input`
`)`
	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).

`load_input ( RowUpdate &$update, array $input, bool $formatted )`	This hook is called when user input is being loaded for either a new record, or a modification to an existing record. Because this input may come via an HTTP, SOAP or JSON request, it is not always correct to look at $_REQUEST (or $_GET or $_POST) for this information.
`load_request ( RowUpdate &$update, array $req, bool $ignore_blank )`	A lower-level hook than `load_input`, this method may be used to capture any form input, including fields which do not correspond with a known (updatable) field from the model, and uploaded files.
`fill_defaults ( RowUpdate &$update )`	Called when a `RowUpdate` object is validated (after the standard validation checks and before saving), this hook should be used to populate fields which have calculated values, often depending on the values of other fields. Doing so in a `before_save` hook is not always sufficient, as required fields may be flagged as missing in that case.
`before_save ( RowUpdate &$update ) after_save ( RowUpdate &$update )`	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 `$updates` property of the `RowUpdate` object. A `before_save` hook may throw an `IAHActionCompleted` exception to indicate that the record update has been completed and the default behaviour must be skipped. An `IAHActionAbort` exception indicates that certain conditions have not been met and the record update cannot be completed.
fill_defaults (RowUpdate &$update)	Called as part of the process for saving a record, this hook is intended to populate required fields which are not provided by the user, but may otherwise be determined.
`validate ( RowUpdate &$update )`	The last step when a `RowUpdate` object is validated before saving. Additional validation checks may be performed on the field values, and validation errors added using `$update->addValidationError('invalid_value', string $field_name)`.
`notify ( RowUpdate &$update )`	This hook is called after a successful `save` operation in order to send notification emails or otherwise alert users to the changes.

`before_delete ( RowUpdate &$update ) after_delete ( RowUpdate &$update )`	These hooks are called when a record is to be deleted (by setting `deleted=1` in the record, not removing it from the table). Like the `before_save` hook, the `before_delete` hook may throw `IAHActionCompleted` or `IAHActionAbort`.
`before_add_link ( RowUpdate &$update, string $link_name ) after_add_link ( RowUpdate &$update, string $link_name )`	These hooks are executed when a record is being added or updated in a `link` model table. The details of the relationship data may be accessed via the `$link_update` property of the `RowUpdate` object. This hook is called for the models on both sides of the relationship. Like the `before_save` hook, the `before_add_link` hook may throw `IAHActionCompleted` or `IAHActionAbort`.
`before_remove_link ( RowUpdate &$update, string $link_name ) after_remove_link ( RowUpdate &$update, string $link_name )`	Called when a relationship between two records is being removed. Like the `before_save` hook, the `before_remove_link` hook may throw `IAHActionCompleted` or `IAHActionAbort`.

There are also a special set of logic hooks associated with the User model, used to perform actions as part of the user's browsing experience:

`after_login ( string $user_id, string $login_type )`	Executed after a successful login, including logins via the SOAP or JSON interfaces.
`page_init ( BasePage &$page )`	This hook are executed when a BasePage is initialized (early in the rendering pipeline for normal web-based sessions). It can be used to inject global javascript libraries or CSS styles, for instance.
`before_page_render ( BasePage &$page ) after_page_render ( BasePage &$page )`	These hooks are executed later in the rendering pipeline for a standard application page, once all normal global variables have been initialized and permissions have been checked for the current action.

4.4.2 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 (these are listed in section 4.4.3). 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.

`type`	The column type, which corresponds indirectly to an SQL column type. See the table of common field types below.
`dbType`	A value overriding the database column type, which is generally inferred based on the `type` value.
`vname`	A reference to a language string in either the module or application language files representing a label for this field.
`vname_list`	A language string to override `vname` in the context of list column labels.
`audited`	A flag indicating that updates to this field are logged to the associated audit table, as long as `audit_enabled` is set in the model detail descriptor.
charset	For varchar-type fields, this property may be set to 'ascii' in order to restrict the input to ANSII characters and reduce the database storage requirements to one byte per character.
`comment`	A string describing the usage of this field.
decimal_places	For `float` or `double`-type fields, the number of decimal places to display in the user interface.
`default`	A default value for the column when none is specified by the user or by one of the pre-save hooks on the model (`fill_defaults` or `before_save`).
detail_link	Set to true in order to render the field as a link to the target record when included in a ListView.
`editable`	Set to false to disable user editing of a field, including on new records.
id_name	For `ref`-type fields, the name of the corresponding ID field. When not provided this will default to the name of the `ref` field with '_id' appended. If not defined explicitly then the corresponding ID field will be automatically created.
`importable`	Generally defaulting to true, set this flag to false to hide this field inside the ImportDB module.
`len`	The length of the corresponding database column in characters.
`massupdate`	A flag to control visibility of this field on the ListView's mass-update panel.
`reportable`	Whether to allow make this field available for reports.

`required`	Marks this field as required, meaning it must contain a non-null value.
`updateable`	Like `editable`, disables user updates to the field, but only for existing records.
`unified_search`	A flag indicating that this field should be added to the default unified search filter.
width	The normal rendering width of the field in characters (if not overridden by the layout) when shown in a ListView, DetailView or EditView.

4.4.3 System-Level Field Descriptors

These field descriptors may be referenced to include standard field descriptors (each an array specifying standard properties for the given field) within a model descriptor file. Properties of the standard field descriptors may be overridden by listing them underneath this key.

`app.id`	A standard record ID.
`app.date_entered`	A `datetime` representing the record creation date.
`app.date_modified`	A `datetime` representing the last modification date.
`app.created_by_user`	The user who created this record (a `ref` field).
`app.assigned_user`	The user assigned to this record (a `ref` field).
`app.modified_user`	The user who last modified this record (a `ref` field).
`app.currency`	A standard currency `ref` field.
`app.exchange_rate`	A standard exchange rate field.

4.4.4 Common Field Types

`id`	A 36-character string field containing a unique, system-generated identifier (GUID).
`varchar`	A string value.
`char`	A string value, defaulting to ASCII (8-bit) database representation.

`text`	This field type represents a multi-line text field and is stored in an SQL `text` column.
`tinyint, int, float, double, currency, base_currency, percentage`	Standard numeric field types.
`bool`	A true or false value, usually represented as an SQL `tinyint`. Fields of this type are rendered as checkboxes.
`date, time, datetime`	Standard date and time field types. These are always stored in GMT, and shown to the user in their local time zone.
`duration`	A duration field, stored as an integer representing a number of a minutes.
`enum`	A dropdown list, usually represented as a `varchar` column and having an associated `options` array. Options may also be defined programmatically by defining an `options_function` property, referencing a function which produces the array of options for the selection input.
`multienum`	A set of values chosen from a multi-select list. In the database values are stored in a single string with '`^,^`' as the separator between values.
`phone`	A phone number, stored using a `varchar` column.
`email`	An email address, stored using a `varchar` column.
`url`	An internet URL, stored using a `varchar` column.
`ref`	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 `id_name` 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 `ref` field defines `bean_name`, representing the name of the target model. Otherwise, a `ref` field must define `dynamic_module` (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.
`html`	An HTML field, such as the body of an email template.
item_number	A simple string value, but generally rendered using fixed-width characters. This field type often used to represent product identifiers and serial numbers, as well as unique numeric IDs for various record types.

module_name	A reference to a module name, used for instance when implementing a `multi-ref` input (which allows both the related module and ID to be selected).
file_ref	A reference to an uploaded file. When rendered, this field type will automatically handle uploading and storage of the associated file.
image_ref	Essentially a `file_ref` field specialized for image-type files.

4.4.5 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. The primary key index is specified automatically (based on the `primary_key` property in the `detail` section of the model) and does not need to be repeated.

A sample index definition used by the EmailTemplate model

```
indices
        idx_email_template_name
                fields
                        - name
```

4.4.6 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 in a separate model descriptor. 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`.

4.5 Localization

In 1CRM 7, the organization of translatable language strings changed significantly in comparison to earlier versions. 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 1CRM 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 and in layout 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 placed in `lang.en_us.lists.php`.

4.6 Model Display Descriptors

In addition to the model descriptor file, most 1CRM model classes will also be associated a display descriptor file. These are located in the `display/` subdirectory of each module. This file is used to define standard filters for the model as well as any non-database fields and various display-related settings. Note that all of these settings are optional.

list.default_order_by	A field name (with optional 'ASC' or 'DESC' appended) representing the default sort order for this model's ListView, overriding the default_order_by defined by the model descriptor.
list.buttons	An array of button descriptors representing mass-update actions on the ListView for this model. Each entry should generally define a `vname` property (the label), an `icon`, and a `perform` property containing javascript to submit the mass-update action (this generally means calling `sListView.sendMassUpdate`). These buttons generally map to mass-update handlers, defined below.
list.massupdate_handlers	An array of mass-update handler descriptors. Each entry is an array defining a few required properties: `name`, the unique name of the mass-update action; `class`, the name of the class which will perform the action; and `file`, the path to the file containing that class. Once the class is loaded, the static class function `listupdate_perform (ListMassUpdate $mu, string $perform, ListFormatter &$list_fmt, ListResult &$list_result, $uids)` is called in order to perform the mass-update action.

`list.layouts`	An array of standard ListView layouts for the model, which will be represented as tabs along the top of the form. Normally each array key represents the name of the layout, but this can be overridden by setting the `view_name` property. Set the `vname` property to provide the tab label. An `override_filters` array may also be provided in order to set default values for ListView filters, whether they are shown on the filter form or not.
	
	An example from the Accounts module, adding a Customers tab to the ListView form:
	`list`
	`layouts`
	`Customers`
	`vname: LBL_CUSTOMERS`
	`override_filters`
	`is_supplier: 0`
	`account_type: Customer`

`list.show_favorites`	Show a favorites column in the ListView and DetailView, along with a standard filter to display only favorite records.
`edit.quick_create.via_ref_input`	Allow the user to quick-create new records when a ref field based on this model is placed on any standard EditView form.
`view.layouts`	Similar to `list.layouts`, this property may contain an array of alternate layouts for the DetailView. Each entry will be represented as a tab at the top of the standard DetailView form.
`basic_filters`	A simple list of ListView filter names which are shown by default in the Browse ListView layout and in popups.
auto_filters	A list of filter names which are to be applied automatically when included in the HTTP request, even when not placed on the current filter form.
filters	See the next section for more information on module filters.

fields	A set of field descriptors, exactly like the field descriptors in the model descriptor file but assumed to be non-database fields (generally widgets or other virtual fields like addresses).
hooks	A set of display hooks associated with the model. These are explained in section 4.6.3.
widgets	Definitions for custom display widgets. See section 4.8 for more information.

4.6.1 ListView Filter Definitions

Each entry in the `filters` section of the display descriptor file defines a separate ListView filter. It is not generally necessary to define filters for existing database fields; instead these filter definitions are used to create more complex restrictions on the ListView results while providing a simple external interface.

There are a few supported filter types. The most basic is the `flag` filter, which is presented as a simple checkbox on the filter form. In each case the `vname` defines the displayed name of the filter, which may be translated. An example definition from the Accounts module:

```
filters
    nonzero
        type: flag
        default_value: false
        vname: LBL_NONZERO_BALANCE
        operator: non_zero
        field: balance
```

In this case ‘nonzero’ defines a flag filter, off by default, which restricts the ListView results to those with a non-zero value for the `balance` column. When the user checks the button labeled ‘Non-Zero Balance Only’, the filter becomes active. Normally flag filters are ignored unless the filter value is set to true, but the `negate_flag` property may also be set in order to reverse this behavior. To test this particular filter, one could pass HTTP parameters `nonzero=1&query=1` in the URI for the ListView form.

The next major filter type is the `section` filter. This is normally rendered as a dropdown list, and presents an set of alternate filter actions to be selected between. An example from the Users module:

```
filters
    status
        type: section
        field: status
        vname: LBL_STATUS
        options_function: [User, get_status_options]
        default_value: NotInactive
        filter_clause_source: [User, get_search_status_where]
```

In this example the options for the dropdown list are provided by a callback (`options_function`), but they could also be written in place using the `options` property, as with an enum field definition. Normally the behaviour of a section filter is simply to restrict the set of records by setting the database column `field` equal to the filter value, but in this case a custom filter clause generator is used to generate the desired expression.

Most basic field types can be automatically used as filters, including `varchar`, `date`, `time`, `ref`, and the various numeric fields. Often the rendering of filter inputs will vary from that on a standard EditView in order to allow for more flexibility.

4.6.2 Display Hooks

In addition to the model hooks which are generally associated with RowUpdate objects, 1CRM supports a set of display hooks which are associated with form generator objects.

`view ( StandardDetailManager $m )`	Executed after a DetailView form has been initialized but before it is rendered.
`edit ( StandardDetailManager $m )`	Executed after an EditView form has been initialized but before it is rendered.
`after_edit ( StandardDetailManager $m )`	Executed after a successful update is performed to a record via an EditView form or Delete button.
`before_subpanel_create ( StandardDetailManager $m, &$stop )`	Called when a new record has been created by the user and is about to be added to a subpanel on the parent record. $stop may be set to a true value in order to prevent the action.
`after_subpanel_create ( StandardDetailManager $m )`	Called after a new record has been saved by a user and added to a subpanel on the parent record.

4.7 Layout Descriptors

The use of HTML templates in 1CRM 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 subdirectories and prefixed with the relevant action name. Custom overrides for layout templates (as generated by the layout editor) are stored in `custom/modules/MODULE/new_views/`.

`view.Standard.php`	The DetailView form layout. Other layouts named as `view.*.php` may be accessed using specific values for the `layout` request parameter (in particular when using a tabbed form layout).
`edit.Standard.php`	The standard EditView form layout.
`list.Standard.php`	The standard ListView column layout. Other layouts named as `list.*.php` 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 standard 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
                --
                    field: number
                    list_position: prefix
                    list_format: separate
            width: 60
        - assigned_user
```

In this example the display of the `number` additional column field is customized using the `list_format` and `list_position` options. The first, `list_position`, may be set to `prefix` or `suffix`, in order to display the value either before or after the primary field value without an additional line break. The `list_format` property defines how the value is stylized: `separate` adds a colon (possibly language dependent) character between the two values; `parenth` wraps the value in parentheses; `brackets` wraps it in square brackets; and `hyphen` adds a hyphen character as a separator. This example would be formatted as one line in the form "`number`: `name`" (possibly wrapping onto multiple lines for long values).

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 title 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). The entry may be a simple string naming the link descriptor, or an array if additional properties of the subpanel are to be customized (including the `vname`).

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 `perform` 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.

4.8 Display Widgets

It will often occur in custom extensions to 1CRM that there is a need to generate HTML outside of the normal HTML form generator. In these cases, and indeed for many cases within the 1CRM system, display widgets are used to encapsulate the rendering and processing logic for custom buttons, form fields, and form sections. These widgets may then be embedded in ListView, DetailView and EditView forms.

Application-level widgets are defined in the system file `include/config/display/display.app_widgets.php`. For widgets which are specific to a single module or which are to be packaged as part of an extension, the `widgets` section of a display descriptor file may be used instead. Widget definitions simply register the widget with the system, along with a unique name, its basic type, and the path to the file containing it:

Sample widget definitions

```
widgets
    PdfButton
        type: form_button
        path: include/layout/widgets/PdfButton.php
    RunIntervalInput
        type: field
        path: include/layout/widgets/RunIntervalInput.php
    SocialAccountsWidget
        type: section
        path: modules/SocialAccounts/widgets/SocialAccountsWidget.php
```

Widgets of type `form_button` must inherit from the FormButton class (`include/layout/forms/FormButton.php`). They can be included in the form_buttons section of a DetailView or EditView layout by simply setting the `widget` property of the button definition to the name of the widget.

Widgets of type `field` can be included either in a ListView layout as a column, or in a DetailView or EditView form as a single cell with an associated label. They must inherit from FormField (`include/layout/forms/FormField.php`) or from a subclass.

The last type, `section` widgets represent an entire table in an DetailView or EditView form. Examples include the social accounts panel on an Account, and the line items editor used in Quotes or Invoices. These widgets must inherit from FormSection (`include/layout/forms/FormSection.php`) or from a subclass such as FormTableSection.

Each widget generally overrides the `renderHtml(HtmlFormGenerator &$gen, RowResult &$row_result, array $parents, array $context)` method in order to perform its rendering. For `field`-type widgets which may be included in a ListView, it may be desirable to override the `renderListCell(ListFormatter &$fmt, ListResult &$result, $row_id, $list_params=null)` method for an alternate rendering format. If certain database fields are required in order to produce the result, then they should be returned in an array from a custom `getRequiredFields()` method.

When included in an EditView form, widgets are also given a chance to respond to certain form events. It may be desirable to override these FormElement methods in order to perform additional processing within the widget class, as opposed to setting separate hooks on the model itself:

```
function loadUpdateRequest(RowUpdate &$update, array $input)
function validateInput(RowUpdate &$update)
function beforeUpdate(RowUpdate &$update)
function afterUpdate(RowUpdate &$update)
```

5.0 Extending System Modules

1CRM provides upgrade-safe methods to extend existing modules.

5.1 The ext/ subdirectory

Each 1CRM module may contain an `ext/` subdirectory providing extensions to the model definition, layouts, or language of another module, as well as similar extensions to application-level configuration files and the 1CRM Administration module. In practice the `ext/` subdirectory is not employed by system modules, and is used exclusively by custom modules. These extensions are indexed by the ExtManager configuration class (`include/config/ExtManager.php`) and cached in `cache/system/ext_cache.php` in order to avoid scanning every subdirectory on each page load.

In general, files located under these directories are parsed by the configuration manager after the base system configuration files (and after the configuration files of any module being extended), but before any site-specific customizations located under the `custom/` subdirectory. This means that custom layouts saved by the layout editor and modifications to the module or application language saved by the dropdown editor will override or extend module extensions.

For the purpose of this document we will assume the existence of a custom module named TestModule which performs several (arbitrary) extensions to the system and to standard system modules. For testing purposes you may wish to create this module under your development system's `modules/` directory. Be sure to delete the extension cache file along with any related caches (model, display, or language) in order to see any changes. When a custom module is installed using the Upgrade Wizard these caches are automatically refreshed.

5.1.1 System Language Extensions

The standard system language files under `include/language/` may be easily extended by creating corresponding configuration extension files under the `ext/include/language/` subdirectory of any module. In particular, the system language strings file may be extended by `ext/include/language/lang.en_us.strings.php`, and the language lists (dropdowns) file by `ext/include/language/lang.en_us.lists.php`. In this example we are making changes to the dropdown list options in the English language file. Any dropdown options not defined by other language packs will be automatically inherited when those language packs are in use.

modules/TestModule/ext/include/language/lang.en_us.lists.php

```
<?php return; /* no output */ ?>

# create a new dropdown options list
test_dropdown_dom
    first: First Option
    second: Second Option

# add an option to an existing list
account_type_dom
    Nemesis: Nemesis

# replace an existing options list using @clear to erase any previous entries
terms_dom
    @clear
    Now: Now
    Never: Never
```

5.1.2 Model and Display Extensions

For any existing module located in `modules/M/` (for instance), a custom module may choose to define override files located under its own `ext/modules/M/` subdirectory. At this time module extensions are limited to model, display, language, and layout modifications.

These extensions all follow the same pattern. They are parsed directly by the configuration manager and operate on the tree structure resulting from existing configuration files.

Extending a model descriptor for an existing module is straightforward. In this example we add a new field to the Accounts module. We use the `vname_module` field parameter so that the language string may be defined in the language/ directory of our custom module. Note that the Database Repair task must be run in order to create the corresponding column in the database:

modules/TestModule/ext/modules/models/bean.Account.php

```
<?php return; /* no output */ ?>

fields
    test_duration
        type: duration
        vname: LBL_TEST_DURATION
        vname_module: TestModule
```

Similarly, we can extend the display model of the Account model. In this example we add a new filter, and automatically place it on the Browse layout of the Accounts ListView by adding a new entry to `basic_filters`.

modules/TestModule/ext/modules/display/display.Account.php

```
<?php return; /* no output */ ?>

basic_filters
    only_customers
filters
    only_customers
        type: flag
        vname: LBL_ONLY_CUSTOMERS
        vname_module: TestModule
        field: account_type
        value: Customer
```

5.1.3 Module Layout Extensions

There are two methods to extend module layouts. Existing layouts for a module `M` may be modified by creating a corresponding file under the `ext/modules/M/views/` subdirectory of the custom module, while new layouts and replacements for existing layouts in said module may be added to `ext/modules/M/new_views/`.

Replacement views are no different from normal layout files, but view extensions may use additional methods in order to place fields at the correct location within an existing layout. Under the `layout` property, view extensions may define the `ext_elements` property in order to list layout modifications with respect to existing fields in the layout. These modifications are able to add new fields, remove existing fields, and replace existing fields by new content. Modifications are made relative to existing fields, which must be named, and must have been previously placed within the layout. Each entry in ext_elements may define either a `name` (for a single field), or an `elements` array. It must define one of the properties `after`, `before`, `replace`, or `remove`.

In this example we update the standard Accounts layout by adding our new custom field (defined in section 5.1.2), then swapping two existing fields (`website` and `email1`).

modules/TestModule/ext/modules/Accounts/views/view.Standard.php

```
<?php return; /* no output */ ?>

layout
    ext_elements
        # add our custom test_duration field after 'name'
        --
            after: name
            name: test_duration
        # remove the email1 field
        --
            remove: email1
        # replace the website field with a pair of fields, email1 and website
        --
            replace: website
            elements
                - email1
                - website
```

Layout extensions may also add or remove subpanels using a similar method, via the `ext_subpanels` property. In this example we add the `cases` subpanel to the Sales layout for the Accounts module, directly after the `opportunities` subpanel. Note that each subpanel must refer to an existing link definition in the corresponding model – to add a new subpanel, a new link definition must also be added using the model extension method outlined in section 5.1.2. Each entry in ext_subpanels may define a `before` or `after` property; otherwise the new subpanel is added to the end of the list.

modules/TestModule/ext/modules/Accounts/views/view.Sales.php

```
<?php return; /* no output */ ?>

layout
    ext_subpanels
        # add the standard cases subpanel after the opportunities subpanel
        --
            after: opportunities
            name: cases
```

5.1.4 Extending the Administration Module

The Administration index page may be easily updated by custom modules. By simply adding files named as `ext/modules/Administration/administration.Custom.php` (the Custom part is arbitrary), additional entries may be added to the `$admin_group_header` array defined in `modules/Administration/index.php`. Each of these files is included and parsed as normal PHP code.

In this example we add a new entry to the first group of links. Note that `modules/TestModule/Configure.php` must be created separately, and must be added to the `web_actions` section of TestModule's module information file (under `metadata/`) in order to be accessed.

modules/TestModule/ext/modules/Administration/administration.Custom.php

```
<?php
$admin_group_header[0][3]['testmodule'] = array(
    'Administration', # the icon
    array('LBL_TESTMODULE_CONFIG', 'TestModule'), # the title
    array('LBL_TESTMODULE_CONFIG_DESC', 'TestModule'), # the description
    './index.php?module=TestModule&action=Configure')
);
?>
```

6.0 Debugging Methods

6.1 Application Settings

When developing it can be helpful to enable one or more settings in the 1CRM configuration. Some of these will have UI equivalents in the Configurator module, but most are hidden and must be added manually to the application config file, `include/config/local_config.php`. These settings are intended for temporary debugging or on private development sites only, as they may pose a security risk or performance overhead on public sites.

`cache.disabled:` Disable the external memory cache (such as APC) if any has been detected.

`site.allow_debug:` When this flag is enabled, ListViews and DetailView forms may have additional debugging information shown by adding `&debug=1` to the page URI. This includes printing the generated SQL queries.

`site.performance.calculate_response_time:` When enabled, 1CRM calculates and displays the time required to render the current page in the footer.

`site.performance.show_page_resources:` Show the number of PHP files included as well as statistics on the external cache.

`site.performance.show_memory_usage:` Show the amount of memory used in preparing the page.

`site.performance.log_all_queries:` Log all database queries performed during the generation of the page to `sql.log`, along with the time required for each.

`site.performance.suppress_display_errors:` Normally, 1CRM disables the PHP configuration setting `display_errors` during initialization. In order to display PHP errors as they arise, set this value to false.
`site.performance.force_display_errors:` Set this value to true in order to display all PHP errors and notices. This has the same effect as `error_reporting(E_ALL); ini_set('display_errors', 1)`. Note that syntax errors raised before system PHP files are included may still result in a blank page, depending on the PHP configuration.

`site.performance.suppress_deprecation_warnings, site.performance.suppress_strict_warnings:` Normally 1CRM suppresses any PHP notices of type E_DEPRECATED or E_STRICT. Set these flags to false in order to log or display them.

`site.log.detail_errors, site.log.detail_internal_errors:` When a PHP exception is raised during page rendering 1CRM will display a simple error message. In order to show a stack trace of the exception instead, enable these settings. `detail_internal_errors` is required in order to display exceptions subclassing `IAHInternalError`, as these are considered more sensitive.

`site.js_custom_version`: Set this property to a different value (generally an incrementing integer) in order to override any javascript caching on the client or server side.

`layout.jsmin_enabled:` Set this value to false in order to disable the built-in javascript caching, which causes javascript files to be minified and loaded via jsmin.php.

`layout.show_validate_button:` This setting enables a Validate setting on all EditView forms, which may be used to check record pre-save conditions and display a detailed report of any issues discovered.

`json.log_level, soap.log_level:` Set these values to a custom error level (info, warn, error, or fatal) in order to log details of all requests to the given external interface (SOAP or JSON) to the application log file.

6.2 Utility Functions

AppConfig::current_user_id()	Fetch the ID of the current user.
AppConfig::setting ($name, $default=null, $standard=false)	Fetch a setting from the application config, returning `$default` if it is not defined. Pass `$standard` to return the application default for a setting, ignoring any custom setting.
$log->info ($msg) $log->warn ($msg) $log->error ($msg) $log->fatal ($msg)	Write a custom message to the 1CRM system log file. Note that if the log level is below the minimum (adjustable on the System Settings page) then it will be ignored.
`pr2 ( $message, $title=null, $wrap=false, $hide=false, $escape=true )`	Use this function to quickly output debugging information. Arrays and objects are automatically formatted to be more legible. Word wrapping is enabled via the `$wrap` parameter, and the output may be shown in a more compact format by setting the `$hide` parameter. The `$escape` parameter (default true) escapes any HTML characters in the message.
`tr2 ( $format=false, $title=null, $html=true, $hide=false )`	This function can be used to quickly print a formatted stack trace and output via the `pr2` function. In order to return the output instead of printing it, pass the format parameter.
`prq ( $query, $title=null, $hide=false )`	Quickly format an SQL query with syntax highlighting and output it via the `pr2` function.

Appendix A - Standard Icons

In addition to the theme icons located in each theme's images/ subdirectory, which are used primary to identify different modules, 1CRM defines a standard set of icons for common actions. When adding buttons and similar UI elements to a layout, it is generally preferable to choose from this set of icons when possible. Icons are rendered by creating a div element of class `input-icon` and adding the specific class (for example `<div class="input-icon icon-accept"></div>`). The following table summarized these icons and their intended uses.

icon-add	Create a new record
icon-accept	Save changes to a record or confirm another action
icon-action	For a standard 'tools'-type menu, shown as a gear icon
icon-cancel	Return from an action without committing any changes
icon-calendar	Used by date and datetime-type fields
icon-changelog	For the change log (audit log) of a particular module
icon-convert	Convert a record, generally by creating a new record in a separate module
icon-close	Close the current form or window
icon-delete	Delete a record
icon-duplicate	Duplicate a record
icon-edit	Edit a record
icon-editlayout	Edit a standard form layout
icon-editlist	Edit a ListView layout
icon-email	Create a new email or link to the Emails module
icon-exchangerate	For actions relating to a record's exchange rate
icon-expand	Expand a record, subpanel, or part of a record
icon-export	Export a record or set of records
icon-filter	Add a filter to a ListView or other object
icon-help	For quick help tips or links to other documentation
icon-layout	To identify form layouts
icon-note	For creating related Note objects or linking to the Note modules

icon-print	Prepare the current record or set of records for printing
icon-reports	Used by the Reports tab on each relevant ListView
icon-return	Return to the previous step or action
icon_search	Search for a record
icon-send	Send an email or other notification
icon-skype	For integration with the Skype messaging system
icon-sortlist	To adjust the sort method of a particular list
icon-sources	For adjusting related sources on a ListView
icon-star	Used to indicate Favorite records
icon-recur	For recurring events or scheduled items
icon-teams	For team restrictions and other links to the SecurityGroups module
icon-time	For time-type fields
icon-user	Used to represent a single, non-administrative user
icon-users	Used to represent a mixed group of users
icon-adminuser	Used to represent an administrative user
icon-view	Show details on a selected record

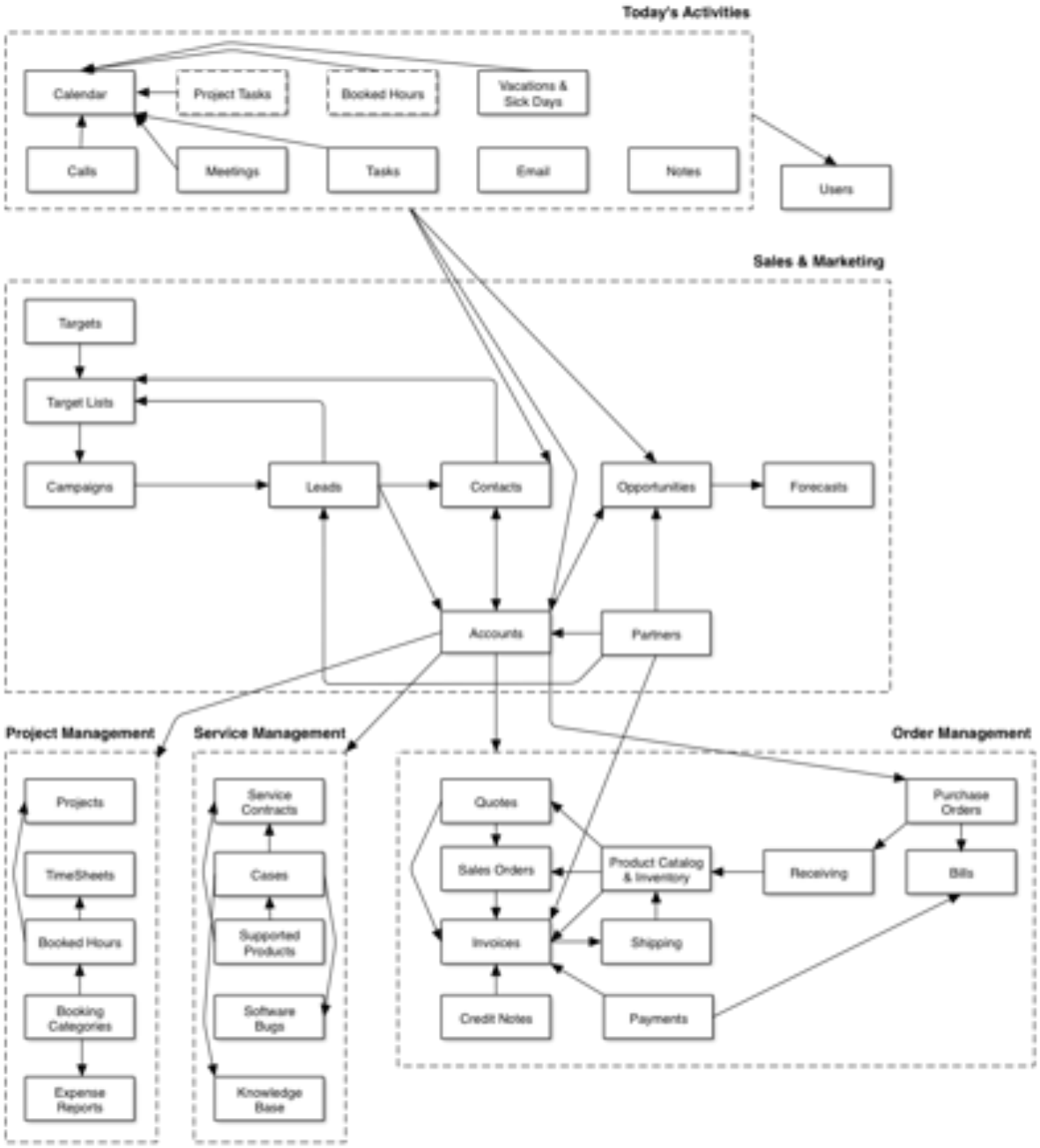
Among these standard icons there are also icons intended for inline use within text or for navigation purposes. These may not always be the correct size for embedding in a standard form button.

icon-info	Show quick information on a record
icon-popup	Indicates a button which produces a popup window
icon-up icon-down icon-left icon-right	Large arrow icons, generally used when moving content within a container. For example, when moving line items within a Quote
icon-prev icon-next icon-start icon-end icon-dprev icon-dnext	Navigation arrows, used to move between records or pages of records

`icon-ledgrey`
`icon-ledgreen`
`icon-ledred`
`icon-ledyellow`
`icon-ledviolet`
`icon-ledblue`
`icon-ledorange`
	Standard LED-type indicators in different colors

`icon-temail`
`icon-tlink`
`icon-tphone`
	Small type indicators for links to email addresses, external web addresses, and phone numbers respectively

Appendix B - High-Level Design



1CRM DEVELOPER GUIDE

A Comprehensive Guide to Developing Customizations and Extensions for 1CRM