CIVICRM DEVELOPER GUIDE

Published : 2011-09-01
License : None

INTRODUCTION

1. INTRODUCTION

This manual contains developer documentation for working with CiviCRM. It is intended as a warm up for developers. It is aimed at both experienced developers who need to know their way around CiviCRM and experienced CiviCRM users who want to extend their skills and customise their systems further. Organizations that want to ensure that their developers are using best practices to extend CiviCRM may also find it useful.

We hope it will help you and encourage you to engage with the CiviCRM community.

CONFIGURING, CUSTOMISING AND EXTENDING

CiviCRM is very flexible. It aims to cater to the majority of non-profit needs through its many configuration options. When you reach the limit of configuration, it is time to start customising and extending. In this manual when we use the word configure, we are referring to something that you can do via the user interface. When we use the words customise and extend, we are referring to writing code.

It is important to remember that there are many techniques (adding fields, profiles, components, Drupal modules and Joomla! extensions) that can be used to configure CiviCRM without writing a line of code. To get the most out of CiviCRM through these techniques, and to avoid unnecessary coding, we recommend you read the CiviCRM user manual and consult with the community before starting any coding project.

Once you are sure that what you want to do cannot be achieved through the user interface, it is time to start writing custom code. So read this manual - it is written for you!

Some of the more popular ways to code with CiviCRM are:

1. Edit the templates. This technique is useful to change the layout of forms, and to add more data or features via AJAX.
2. Write code that implement hooks. Hooks are called at specific points in the code as users interact with CiviCRM and are useful in many different circumstances, for example
    1. to change the default behaviour when you create, update or delete an entity
    2. create a new custom mail merge token
    3. populate a custom field automatically based on your own business logic.
3. Write and package an extension. Useful as a technique to add a custom search, report, payment processors, etc.
4. Override and/or patch existing code. Small changes and improvements to the way in which CiviCRM works can be achieved by editing the existing code. You should definitely chat with the developer community before embarking on any code that changes existing code and you should submit a patch to CiviCRM if the code will benefit others.
5. Write a new CiviCRM component. This is a major undertaking and only recommended for the seasoned CviCRM developer. You'll definitely want to discuss and collaborate with other CiviCRM developers if this is the path you choose.

Regardless of your chosen method, chances are that someone else will be interested in what you are doing. Right from the start, you should talk about - and share - your ideas with the community.

2. BEFORE YOU START

Before you start on any code to extend CiviCRM, it is really important to discuss your ideas with the community. Here are a few of the reasons why this is a good idea:

- It may have been done already
- You'll get suggestions and advice on suitable ways to approach the problem
- Other people will know what you are doing, and be able to contact you if they want to collaborate

A typical pre-development workflow will start with a discussion on the forum about what you want to do. Here you'll receive feedback from other members of the community and core team about your ideas. You might be lucky and find out that there is a already a way to do what you want using the user interface (and that no coding is necessary). Or it might be that someone has done something similar already and all that is required is a few tweaks to their code.

If and when you have confirmed that some coding is required, it is good practice, even for relatively small projects, to write

- a requirements document which describes in detail what you want the finished code to do
- a specification that outlines how you are going to meet these requirements with CiviCRM

The requirements are typically written to be understandable to end users, and the specification can be thought of as a translation of those requirements into the language of CiviCRM. Both requirements and specification should go on the wiki.

Once you've written the requirements and specification document, you should go about soliciting feedback. Get feedback on the requirements from your end users and feedback on the requirements and the specification from anyone that is willing to comment. To encourage more discussion, you can write a post on CiviCRM's blog, tweet it out with the #civicrm hashtag, tell similar CiviCRM users and organisations and so on. The more feedback you can get the better.

If you get stuck writing your requirements and specification, or would like to get more background, have a look at some requirements and specifications written by others - there are plenty on the wiki.

MAKE IT HAPPEN

If you need or would like extra resources for your code improvement, you should consider a 'make it happen' (MIH for short).

Make it happen is a crowd-sourcing initiative for CiviCRM, which incidentally, is built using CiviCRM. Around 15 MIH's were selected for the 4.0 release, and more Make it Happens are likely to be selected for future releases. MIH work is carried out by the core team or trusted consultants. You can see a list of current MIH online at http://civicrm.org/mih. If you think your project would make a good MIH, discuss it with the core team.

3. THE DEVELOPER COMMUNITY

Like many open source projects, CiviCRM is shaped, guided, and driven by its community, a far-flung ecosystem of users, developers, and implementers who utilize CiviCRM in many different ways and bring to it a wide array of skills, experiences, and perspectives. As might be expected of a web-based software project, much of the CiviCRM community's activity occurs online. English is the predominant language for discussion and contributions.

WHERE DO DEVELOPERS HANG OUT?

There are a few places to find developers online. Top four are:

- The wiki - http://wiki.civicrm.org
- The forum - http://forum.civicrm.org/
- IRC - #civicrm on irc.freenode.net
- blogs - http://civicrm.org/blog/

The wiki is a gold mine of useful information for developers. What it lacks in coherence, it makes up for in content. Think of it and treat it as CiviCRM's collective notepad. A place for recipes, new ideas and to do lists to be jotted down without too much attention to getting everything in exactly the right place or completely perfect.

The forum is also a great place for developers. Here you'll find discussions on a range of topics from both developers and from users. It is a great place to find answers to the millions of questions that have already been asked about CiviCRM, and to ask ones that haven't been asked already.

When looking on the wiki and the forum, bear in mind that content does go out of date. If the discussion or article you are looking is dated a while ago, you'd be well advised to check that the there isn't a new approach to the problem.

IRC is a great place to go for any technical questions. Here, with a bit of luck, you'll often find an instant answer to your question or a pointer in the right direction. Definitely go here if you feel like you are banging your head against a wall. Writing your question down in itself can often help you find the answer and if not then perhaps someone else will understand your issue and offer to help.

If you follow this path and no one answers don't be offended. If no one answers your question they may be away from their computer or busy. As with all online communities a little bit of patience and understanding goes a long way.

The CiviCRM blog (http://civicrm.org/blog) is another good source of information and discussion. Blog posts are written by both the CiviCRM core team and other community members and cover a wide range of topics.

A fair amount of developer discussion also occurs outside of these official channels. Using your favorite engine to search for CiviCRM will turn up many articles and posts from other folks' websites and blogs. The CiviCRM team is good at keeping an eye out for these posts and often publicize them through Twitter. To keep abreast of the stream of comments, follow @civicrm and find CiviCRM tweets and tag your own tweets with the #civicrm hashtag.

Additionally you may wish to look at the Jira bug reporting system which contains useful notes on issues past and present. JIRA provides issue tracking and project tracking for software development teams to improve code quality and the speed of development. This is where you should log bugs but it can also be a useful guide when starting to investigate problems.

Log on to Jira, and the the wiki using your civicrm.org login. If you would like your account to be upgraded so you can blog please email info@civicrm.org.

You'll hopefully find these CiviCRM resources to be great sources of help. Everyone who interacts online is encouraged to give back to the community by responding to questions and requests for help and contributing their own ideas and feedback to the conversations. Simply asking your own questions is also a significant contribution to the community. It's likely that someone else is having the same problems or wondering the same thing, and the responses you solicit help build the community's knowledge base.

HOOKING IN OFFLINE

Though the online community is both accessible and active, participating in the CiviCRM community offline can be even more rewarding and can help you connect with others in your area who are developing, implementing, and using CiviCRM.

Many cities and regions hold CiviCRM meet-ups where people gather to learn about CiviCRM, share new ideas, developments, and use cases, and meet other folks involved with the community. You can find out more about meet-ups at http://civicrm.org. Some meet-up and local user groups (or LUGs) also maintain discussion boards at http://civicrm.org/groups. Contact the CiviCRM crew if you'd like a discussion board for your own group on the site.

CiviCRM core team and others also conduct developer training in cities around the globe. Check out http://civicrm.org for info about upcoming trainings and contact CiviCRM if you'd like to host meetups or trainings in your own area.

4. FIXING BUGS

So you've found a bug and you want to fix it. What next? The first thing you should do is congratulate yourself on stepping up to fix the bug. The second thing you should do is to make sure that you are familiar with the content of this chapter so that you can fix the bug in the best possible way.

The three things that make up a great bug fix are:

1. a detailed bug report
2. a patch file
3. tests

WRITING GOOD BUG REPORTS

The best bug reports give lots of background and context. Don't forget that the way you are using CiviCRM is most likely very specific to your organisation. The more background you can give on the bug, the better.

The best bug reports clearly state

- What you did
- What you expected to happen
- What actually happened

CREATING PATCHES

A patch is a text based file format that shows the difference between one or more files. Once you have submitted your patch, CiviCRM core developers can apply it to the development version of CiviCRM.

You should create your patch against the appropriate version of CiviCRM. If you are in doubt as to which version this is, there'll likely be someone on IRC that can give you a immediate answer. Otherwise ask on the forum.

The standard way to create a file is by using the diff command.

```
$diff /path/to/old/file.php /path/to/new/file.php > changes.patch
```

The above will create a patch called changes.patch that you can attach to your issue.

There are cleverer techniques that you can use to generate patches if you are using version control systems.

ACCESS TO SVN

SVN is a free/open source version control system. It manages the changes made to the source code by multiple developers. SVN allows you to recover older versions, examine the history of how the code has changed, and receive and publish changes.

If you are submitting a lot of changes (patches), feel free to ask on IRC for an SVN account so you can directly apply your code contributions.

ADDING TESTS TO BUG REPORTS

US Federal law prohibits the submission of a patch to the CiviCRM code base without a suite of tests that have been shown to demonstrate the correct functioning of the code. It is also a contravention of European Union directive E842317 and enshrined in the UN universal declaration on human rights Article 31. Well not quite! But you might think so if you asked the CiviCRM community, and for good reasons.

Clever CiviCRM developers know that it is in their interest to submit one or more tests with their bug reports. There are at least three important reasons to submit a comprehensive set of tests with your patch.

- it minimises the chances that you will be bitten by the same bug in the future as CiviCRM developers are committed to having the test suite complete successfully before each point release.
- it providers developers with an easy way to find out whether they have actually fixed your bug
- they are often the best way to explain complex user interface bugs.

5. THE CODEBASE

Having a general overview of the codebase and high level understanding of how it is organised, where to find specific types and bits of code, etc. is really useful when you are getting started with developing CiviCRM. This chapter looks at a few key parts of the code base to give you an idea of how everything fits together.

Download CiviCRM from sourceforge.net unzip it and have a look around the codebase. The rest of this chapter explores the directories inside the CiviCRM zip file.

OBJECT ORIENTED

The CiviCRM codebase is object oriented. If you aren't familiar with object oriented programming, spend a little time reading some beginner tutorials on the net. Object orientated programming might be a little intimidating at first, but isn't actually that complicated when it comes down to it, and it is very useful.

BUSINESS LOGIC

Most of the business logic of CiviCRM, that is to say the part of CiviCRM that defines what it does and how it behaves (e.g. that allows people to register on events) is found in the **CRM directory** ('CIVICRM_ROOT/CRM'). Looking in this directory, you'll find directories for core CiviCRM functionality like contacts, groups, profiles, deduping, importing, and the CiviCRM components like CiviCampaign, CiviMail, Pledges, etc.

Each of these directories is slightly different depending on what they do but there are some subdirectories that you'll find in most of them, for example BAO, DAO, Form and Page. So what can you expect to find in these directories?

DAO

DAO stands for data access object. Code in this directory exposes the contents of the database. The DAOs are automatically generated for each release based on the data schema (which is why you won't see them - and will have to generate them - if you checkout CiviCRM from SVN). DAO objects tend to be instantiated in BAO classes.

The DAO has a property for each field (using the actual field name, not the unique name). They also have standard create retreive update delete type functions, etc.

BAO

BAO stands for business access object. BAOs map to DAOs and extend them with the business logic of CiviCRM. A lot of the meat of CiviCRM is found in the BAOs, for example they have the code that creates follow up activities when an activity is created, or create activities and populating custom fields when a pledge is created.

Form

In general each page in CiviCRM which contains a form that maps to a file in one of the form directories. Form files contain a class that extends CRM_Core_Form. This class has different methods that are used before the form is displayed to do things like check permissions, retreive information (preProcess), display the form (buildForm), validate the form (formRule) and carry out tasks once the form is submitted (postProcess). Forms can take information from the BAO to be displayed to users and call the BAO on submission. In general, each form has an associated template (see below) which is used to create the html of the form. Perhaps the best way to get to grips with the Forms is by experience and experimentation.

Page

If a CiviCRM screen isn't a Form, it is probably a page. Pages files contain a class that extend CRM_Core_Page. Similar to the form class, Pages have methods that are called before the page is displayed to control access, set the title, etc. (preProcess), and when the page is displayed (run). Pages tend to take information from the BAO to be displayed to users. In general, each page has an associated template (see below) which is used to create the html of the page.

xml

This directory contains a menu directory which maps urls to CRM form or page classes and controls access to these URLs using permissions.

TEMPLATES

The templates directory contains all the HTML for pages and forms. Directly inside the templates directory is a CRM directory. In general, all templates map to a form or page class. CiviCRM chooses a template for the form or page based on the class name.

For example, the class CRM_Member_Form_MembershipRenewal looks for a template in templates/CRM/Member/Form/MembershipRenewal.tpl

Templates are written in smarty, a common PHP template engine. Variables can be passed to smarty using the assign() method which is available to all Form and Page classes.

Customising templates is discussed in more detail in 'Techniques'

THE API

The application programming interface (API) is stored in the api root directory. Best practice for using the API is discussed in more detail in 'Techniques'

BIN SCRIPTS

The bin directory contains a variety of scripts that can be run to carry out specific functions. Some of these scripts are run on a regular basis, for example the CiviMail 'send' and 'process' scripts. Others are run on a one of or occasional basis, e.g. update geo-coding.

SQL

The SQL directory is automatically generated as part of a release. It contains useful files like the SQL to create the database and insert demo data. Most developers won't need to edit files in this directory.

L10N

This directory contains lots of automatically generated localisation files. You should not need to edit this directory directly. You should instead use CiviCRM's online translation tool transifex.

PACKAGES

CiviCRM makes use of a lot of 3rd party packages for things like the database, form, javascript and pdf libraries, wysiwyg editors and so on. You shouldn't need to edit these packages directory.

DATABASE STRUCTURE

The database structure is defined in a series of XML files. These files are not packaged in the releases but are available in the SVN repository. They are located in Civicrm/xml/Schema. All the folders within this directory also have folders in the main CRM folder which contain a DAO folder and generally a BAO folder too.

Looking in CiviCRM/xml/Schema/Pledge we see 4 files:

- files.xml
- Pledge.xml
- PledgePayment.xml
- PledgeBlock.xml

files.xml is just a list of the other files. Each of the others represents a table in the Database. The XML files describe the database and are used to build both the DAO files and the new database sql generation script.

The XML describes fields, foreign keys and indexes, an example of a field definition is:

```
<field>
      <name>amount</name>
      <uniqueName>pledge_amount</uniqueName>
      <title>Total Pledged</title>
      <type>decimal</type>
      <required>true</required>
      <import>true</import>
      <comment>Total pledged amount.</comment>
      <add>2.1</add>
  </field>
```

WHEN SHOULD I EDIT CORE CIVICRM?

The above should have given you a reasonable idea of what the CiviCRM code based looks like. Remember that most of the time, editing the core code base directly is not the recommended way for developers to customise and extend CiviCRM.

There are other recommended ways for the majority of scenarios - for example the APIs and hooks. Be sure that any edits that you make to the core code base are really necessary.

To help you decide, here are a couple of principles:

- Bug fixes should always be applied to core.
- Some (but not all) new features make sense to put in core, especially if they would be useful to others.

If you aren't familiar with CiviCRM, by far the best way to get a sense if you should be editing core is by talking with the CiviCRM developer community on IRC, the forums, etc.

DEVELOPMENT ENVIRONMENT
6. BASIC SET UP
7. DEBUGGING
8. TESTING

6. BASIC SET UP

When writing CiviCRM customizations and extensions make sure you don't do it in a production environment! You don't want to interrupt your organisations or users' work or corrupt the data in the database. Instead you should always create a separate local development environment where you can do whatever you want to CiviCRM without disturbing anyone else.

If your computer runs Linux or Mac OS X, running CiviCRM on your local machine is pretty easy. If your computer runs Windows, you have a little more work to do, but it's not impossible - you could consider running Linux inside a virtual machine. There's a free program called VirtualBox that makes this pretty easy. We recommend running the latest version of Ubuntu Linux inside the virtual machine. You can download that here: http://www.ubuntu.com/

INSTALLING CIVICRM

http://wiki.civicrm.org/confluence/display/CRMDOC40/Installation+and+Upgrades outlines pre-installation requirements and gives step by step instructions for installing CiviCRM.

INSTALLING FROM SVN

CiviCRM developers use SVN to collaborate. If you are doing a lot of development in collaboration with the core team or other developers, you might want to install from SVN.

Instructions on installing from SVN are available here:http://wiki.civicrm.org/confluence/display/CRMDOC40/Installing+CiviCRM+from+subversion+%28SVN%29+repository

USING SOURCE CONTROL MANAGEMENT

It is good practice to put any code customisations that you write under version control. Keeping your code in a repository means that you have access to all previous versions of the code and can revert back if anything goes wrong (which with any complex project, will inevitably happen one day). Although CiviCRM uses SVN, we recommend you consider using Git for any code that you write. CiviCRM will migrate to Git at some point in the future.

Putting your source code in a public repository is even better as it makes it public from day one and easy to share.

LOGGING EMAILS TO A FILE

If you have a local test environment set up and want to test processes that involve sending emails, you can set CiviCRM up to log emails to a file rather than sending them. To do this, modify your CiviCRM settings (http://civicrm.settings.php) and add following line:

```
define('CIVICRM_MAIL_LOG', '/path/to/mail.log');
```

7. DEBUGGING

When your code isn't doing what you want it to do, it's time to debug. There are lots of options for debugging and there is lots you can do without setting up a sophisticated debugging environment. This chapter contains some simple debugging tips and tricks to get you started and also instructions on setting up XDebug, which is the recommended debugging tool for CiviCRM when you have bugs which you are finding it really hard to squish.

PRINTING PHP VARIABLES

CRM_Core_Error::debug($name, $variable = null, $log = true, $html = true); can be called to print any variable. It is a wrapper around print_r($variable); which does some clever stuff, but print_r($variable); is often all you need.

Following your print_r(); with and exit; to stop the script execution at that point is often useful or necessary.

DEBUG MODE

CiviCRM has a debug mode which you can enable to give you quick access to a couple of useful diagnostic tools, including access to all the smarty variables that make up a page, and access to the backtrace (read more about backtrace below). It also provides shortcut methods to empty the file based cache and session variables.

These tools are activated by adding parameters to the URL that makes up the page, e.g. &backtrace=1. Go to **Admin > Global config > Debugging** to enable debug mode and read more about the available options.

PRINTING SMARTY VARIABLES

When debug mode is enabled, you can display all the smarty variables that have been added to the page by adding &smartyDebug=1 to any URL.

BACKTRACE

A backtrace is a list of all the functions that were run in the execution of the page, and the php files that contain these functions. It can be really useful in understanding the path that was taken through code, what gets executed where, etc.

If display backtrace is enabled in Debugging options then the backtrace will be displayed when an error occurs.

You can also force the backtrace to be printed at any point in the code by adding a call to "CRM_Core_Error::backtrace();"

CLEARING THE CACHE

Using Drupal, you can clear all caches with the **drush** command civicrm-cache-clear.

Alternatively, you can call the following two methods:

- CRM_Core_Config::clearDBCache();
- CRM_Core_Config::cleanup();

which clear the database and file cache respectively.

CHECK THE QUERIES FIRED BY DATAOBJECT

```
define( 'CIVICRM_DAO_DEBUG', 1 );
```

XDEBUG

XDebug is our main recommendation for developers that want to go into hardcore debugging. Readers familiar with what a debugger is and how it works should feel free to skip ahead to the "Setting Up XDebug" section.

What is a debugger?

A debugger is a software program that watches your code while it executes and allows you to inspect, interrupt, and step through the code. That means you can stop the execution right before a critical juncture (for example, where something is crashing or producing bad data) and look at the values of all the variables and objects to make sure they are what you expect them to be. You can then step through the execution one line at a time and see exactly where and why things break down. It's no exaggeration to say that a debugger is a developer's best friend. It will save you countless hours of beating your head against your desk while you insert print statements everywhere to track down an elusive bug.

Debugging in PHP is a bit tricky because your code is actually running inside the PHP interpreter, which is itself (usually) running inside a web server. This web server may or may not be on the same machine where you're writing your code. If you're running your CiviCRM development instance on a separate server, you need a debugger that can communicate with you over the network. Luckily such a clever creature already exists: XDebug.

Setting Up XDebug

XDebug isn't the only PHP debugger, but it's the one we recommend for CiviCRM debugging.

The instructions for downloading and installing XDebug are here: http://xdebug.org/docs/install

Those instructions are a bit complex, however. There is a far simpler way to install it if you happen to be running one of the operating systems listed here.

Debian / Ubuntu Linux

```
sudo apt-get install php5-xdebug
```

Red Hat / CentOS Linux

```
sudo yum install php-pecl* php-devel php-pear
sudo pecl install Xdebug
```

Mac OS X

```
sudo port install php5-xdebug
```

Next Step for All Operating System

Tell XDebug to start automatically (don't do this on a production server!) by adding the following two lines to your php.ini file (your php.ini file is a php configuration file which is found somewhere on your server. Calling the phpinfo() function is probably the easiest way to tell you where this file is in your case.

```
xdebug.remote_enable = On
xdebug.remote_autostart = 1
```

Once XDebug is installed and enabled in your PHP configuration, you'll need to restart your web server.

Installing an XDebug Front-End

After you have XDebug running on your PHP web server, you need to install a front-end to actually see what it is telling you about your code. There are a few different options available depending on what operating system you use:

All Operating Systems

NetBeans is a heavyweight Java IDE (Integrated Development Environment). It offers lots of features, but isn't exactly small or fast. However, it is very good at interactive debugging with XDebug. And since it's written in Java, it should run on any operating system you want to run it on. You can find it at http://www.netbeans.org/

After installing NetBeans, open your local CiviCRM installation in NetBeans and click the Debug button on the toolbar. It will fire up your web browser and start the debugger on CiviCRM. You may went to set a breakpoint in CRM/Core/Invoke.php to make the debugger pause there. For more information, see the NetBeans debugging documentation.

Mac OS X

A much lighter-weight option for Mac users is a program called MacGDBp. You can download it here: http://www.bluestatic.org/software/macgdbp/

After installing MacGDBp, launch it and make sure it says "Connecting" at the bottom in the status bar. If it doesn't, click the green "On" button in the upper-right corner to enable it. The next time you access CiviCRM, the web browser will appear to hang. If you click on MacGDBp, you'll see that it has stopped on the first line of code in CiviCRM. From there you can step through the code to the part you're interested in. But it's probably a better idea to set a breakpoint in the part of the code you're interested in stopping at. See the MacGDBp documentation for more information.

8. TESTING

CiviCRM is a complex piece of software which is constantly evolving through the contributions of many people. As you can imagine, making sure everything is working correctly (i.e. making sure everything is bug free) is not a trivial task. Tests are one method that CiviCRM uses to ensure that things are working as expected.

There are already a great number of tests in CiviCRM, and the number of tests is growing all the time. All tests are collected together in a test suite which is run periodically on the development version of the code base. The results of tests are available at http://tests.dev.civicrm.org/.

During CiviCRM development, if someone introduces a bug in the code for which there is a test, that test will fail. The bug can then be fixed before the next stable release.

WRITING TESTS

All developers and users of CiviCRM are strongly encouraged to write tests to which have a direct impact on the quality of CiviCRM.

There are two types of test in CiviCRM: code tests, and web tests, and we use two different tools for each type of test.

To ensure consistency all tests should be carried out using the standard CiviCRM example data.

CODE TESTS

Code tests ensure that CiviCRM code is working as expected, for example that a contact is created when you call the 'contact create' API method, or a contribution type is retrieved as expected when you call the contribution type BAO. Code tests are written with PHP Unit which is a PHP based testing framework.

PHP Unit tests help you understand your own code better and force you to think about how your code will be used before you write any of it. If you submit unit tests with patches you guarantee that your patch will continue to work as expected and you will make the core team very happy.

There is good documentation on writing PHP unit tests at http://www.phpunit.de/manual/3.2/en/database.html and the wiki has a growing page of example code at http://wiki.civicrm.org/confluence/display/CRM/Writing+a+PHPUnit+testcase+HOWTO

WEB TESTS

Web tests are designed to ensure the interface is working as expected, for example, that the right things happen when you click on the right buttons. Examples of web tests include that the event confirmation screen is displayed when I hit the register for an event button, or that all 23 contacts are displayed when I search for contacts that live in France (in an instance of CiviCRM that has 23 contacts living in France).

You can record tests using the Selenium IDE which you can download from the Selenium website http://seleniumhq.org/. Web tests should be recorded using an instance of CiviCRM that has standard sample data. The demo (running the latest stable version of CiviCRM) and sandbox (running the latest development version) are good candidates to use for writing selenium tests because they have the demo data installed.

SETTING UP A LOCAL TEST ENVIRONMENT

If you want to run tests locally, you need to set up a testing environment. If you want to submit PHP unit tests you will need to code these locally first. Note that you don't need a local testing environment to record selenium test.

To create a test environment, start with a local development environment and then follow the instructions here:
http://wiki.civicrm.org/confluence/display/CRM/Setting+up+your+personal+testing+sandbox+HOWTO

TECHNIQUES
9. Hooks

10. Developing Page Templates
11. The API
12. Importing data

9. HOOKS

Hooks are a common way to extend systems. The way they work in CiviCRM is that, at key points in processing - such as saving a change or displaying a page - CiviCRM checks to see whether you've "hooked in" some custom code, and runs any valid code it finds.

For example, let's say you want to send an email message to someone in your organization every time a contact in a particular group is edited. Hooks allow you to do this by defining a function with a specific name and adding it to your organisation's CiviCRM installation. The name of the function indicates the point at which CiviCRM should call it. CiviCRM looks for appropriate function names and calls the functions whenever it performs the indicated operations.

Hooks are a powerful way to extend CiviCRM's functionality, incorporate additional business logic, and even integrate CiviCRM with external systems. Many CiviCRM developers find themselves using them in nearly every customization project.

A good test for whether or not to use a hook is to ask yourself whether what you're trying to do can be expressed with a sentence like this: "I want *X* to happen every time someone does *Y*."

HOW TO USE HOOKS

How you use hooks depends on whether you're using CiviCRM with Drupal or Joomla!.

Using hooks with Drupal

Check the CiviCRM wiki page for the most up-to-date information on setting up hooks with Drupal: **http://tiny.booki.cc/?hooks-in-drupal**
http://wiki.civicrm.org/confluence/display/CRMDOC/CiviCRM+hook+specification#CiviCRMhookspecification-Proceduresforimplementinghooks%28forDrupal%29

In order to start using hooks with a Drupal-based CiviCRM installation, you or your administrator needs to do the following:

1. Create a file with the extension *.info* (for instance, *myhooks.info*) containing the following lines. Replace the example text in the first 2 lines with something appropriate for your organization:

```
name = My Organization's Hooks
description = Module containing the CiviCRM hooks for my organization
dependencies[] = civicrm
package = CiviCRM
core = 6.x
version = 1.0
```

2. Create a new file with the extension *.module* (for instance, *myhooks.module*) to hold your PHP functions.
3. Upload both the *.info* and *.module* files to the server running CiviCRM, creating a new directory for them under */sites/all/modules* (for instance, */sites/all/modules/myhooks/*) inside your Drupal installation. The directory name you create should be short and contain only lowercase letters, digits, and underlines without spaces.
4. Enable your new hooks module through the Drupal administration page.

Using hooks with Joomla!

Check the CiviCRM wiki for the most up-to-date information on setting up hooks with Joomla!: **http://tiny.booki.cc/?hooks-in-joomla** (http://wiki.civicrm.org/confluence/display/CRMDOC/CiviCRM+hook+specification#CiviCRMhookspecification-Proceduresforimplementinghooks%28forJoomla%29)

In order to use hooks with Joomla!, you or your administrator needs to do the following:

1. Create a file named *civicrmHooks.php* to contain your functions.
2. Create a new directory anywhere on your server outside of your Joomla! / CiviCRM installation (so that upgrades will leave your hooks alone). */var/www/civicrm_hooks* might be a good choice.
3. Upload the *civicrmHooks.php* file to the directory you just created.
4. Go to: CiviCRM Administer > Global Settings > Directories. Change the value of **Custom PHP Path Directory** to the absolute path to the new directory (e.g., "/var/www/civicrm_hooks" if you used that suggestion in the earlier step).

Refine what you want to act upon

When you create a hook, it will be called for all the types of entities. For instance, a civicrm_post is called after the creation or modification of any object of any type (contact, tag, group, activity, etc.). But usually, you want to launch an action only for a specific type of entity.

So a hook generally starts with a test on the type of entity or type of action. For instance, if you want to act only when a new individual contact is created or modified(does this match the code?), start your civicrm_post hook with:

```
if ($objectName != "Individual" || $op != "edit") {
  return;
}
```

On the other hand, if you want to run multiple hooks within the same function, you don't want to return from any single hook. Instead, you can nest the entire code for each hook within your test:

```
if ($objectName == "Individual" && $op == "edit") {
  // Your hook
}
```

Pitfalls of hooks

Because you have little control over what CiviCRM passes to your hook function, it is very helpful to look inside those objects (especially **$objectRef**) to make sure you're getting what you expect. A good debugger is indispensable here. See the Developer Tips & Tricks chapter at the end of this section for more information on setting up a debugger for your development environment.

EXAMPLES OF USING HOOKS

Some example hooks follow. Consult the hooks reference documentation on the CiviCRM wiki to see the full extent of what you can do:
http://wiki.civicrm.org/confluence/display/CRMDOC/CiviCRM+hook+specification

In all of these examples, you'll put the code we provide into your *myhooks.module* file if using Drupal, or the *civicrmHooks.php* file if using Joomla!. Be sure to upload the file after each change to the appropriate location on your server to see the new code take effect.

Additionally, if you are using Drupal and add a new hook to an existing module, you will need to clear the cache for the hook to start operating. One way of doing this is by visiting the page Admin > Build > Modules.

Sending an email message when a contact in a particular group is edited

In order to have CiviCRM tell you when a contact is edited, define the civicrm_pre hook. This lets you see the incoming edits as well as the values of the existing record, because you may want to include that information in the email.

If you are using Drupal, create a function named *myhooks*_**civicrm_pre**. If using Joomla!, create a function named **joomla_civicrm_pre**. We'll assume you're using Drupal for the rest of the example, so please adjust the code accordingly if you're using Joomla! instead.

```
<?php

function myhooks_civicrm_pre( $op, $objectName, $objectId, &$objectRef ) {
    # configuration stuff
    $theGroupId = 1; # group id we want the contacts to be in
    $emailRecipient = 'johndoe@example.org'; # person to e-mail

    # Make sure we just saved an Individual contact and that it was edited
    if ($objectName == "Individual" && $op == "edit") {

        # Now see if it's in the particular group we're interested in
        require_once 'api/v2/GroupContact.php';
        $params = array('contact_id' => $objectId);
        $groups = civicrm_group_contact_get( $params );
        $found = false;
        foreach ($groups as $group) {
            if ($group['group_id'] == $theGroupId) {
                $found = true;
            }
        }
        # Exit now if contact wasn't in the group we wanted
        if (! $found) {
            return;
        }

        # We found the contact in the group we wanted, send the e-mail
        $emailSubject = "Contact was edited";
        $emailBody = "Someone edited contactId $objectId\n";
        # Here's where you may want to iterate over the fields
        # and compare them so you can report on what has changed.
        mail( $emailRecipient, $emailSubject, $emailBody );
    }
```

```
}
```

Validating a new contribution against custom business rules

If you have experience with other hook-based systems, you might think that the civicrm_pre hook is the one to use for validations. But this is not the case in CiviCRM because, even though the civicrm_pre hook is called before the record is saved to the database, you cannot abort the action from this hook.

This is where validation hooks come in. When you return true from a validation hook, CiviCRM saves the new or updated record. When you return an error object instead, CiviCRM aborts the operation and reports your error to the user.

An example follows of using a validation hook to validate new contributions against a business rule that says campaign contributions must have a source associated with them. In this example, we'll assume you are using Joomla!, so if you are using Drupal instead, be sure to change the function name accordingly.

```
<?php

function joomla_civicrm_validate( $formName, &$fields, &$files, &$form ) {
    # configuration stuff
    $campaignContributionTypeId = 3; # adjust for your site if different

    $errors = array();

    # $formName will be set to the class name of the form that was posted
    if ($formName == 'CRM_Contribute_Form_Contribution') {
        require_once 'CRM/Utils/Array.php';
        $contributionTypeId = CRM_Utils_Array::value( 'contribution_type_id',
            $fields );
        if ($contributionTypeId == $campaignContributionTypeId) {
            # see if the source field is blank or not
            $source = CRM_Utils_Array::value( 'source', $fields );
            if (strlen( $source ) > 0) {
                # tell CiviCRM to proceed with saving the contribution
                return true;
            } else {
                # source is blank, bzzzzzzzzzzzt!
                # assign the error to a key corresponding to the field name
                $errors['source'] =
        "Source must contain the campaign identifier for campaign contributions";
                return $errors;
            }
        } else {
            # not a campaign contribution, let it through
            return true;
        }
    }
}
```

Automatically filling custom field values based on custom business logic

This example uses a hook to write some data back to CiviCRM. You can make a custom field read-only and then set its value from a hook. This is very handy for storing and displaying data that are derived from other attributes of a record based on custom business logic.

For example, let's say you are storing employee records and you want to auto-generate their network login account when new employees are added. By doing it in your code, you can enforce a policy for login account names. For this example, let's say the policy is first initial + last name. So if your name is Jack Frost, your network login name would be *jfrost*.

Add a new read-only custom field to CiviCRM called "Network Login" and then find its ID. You can find it either by:

- Checking the civicrm_custom_field table in your CiviCRM database.
- Editing a contact and check the name of the Network Login field.

The code must refer to the ID as **custom_***id*. So if you find that the id of the new field is 74, refer to is as **custom_74** in your code.

Now that we have our Network Login field, let's see how to populate it automatically with a hook. We'll switch back to the Drupal naming convention for this example.

Note that we use the civicrm_post hook here because we need the new contact record's ID in order to save a value to one of its custom fields. New records don't have an ID until they have been saved in the database, so if we ran this code in the civicrm_pre hook, it would fail.

```
<?php

function myhooks_civicrm_post( $op, $objectName, $objectId, &$objectRef ) {
    # configuration stuff
    $customId = 74;

    if ($objectName == 'Individual' && $op == 'create') {
        # generate the login
        $firstName = $objectRef->first_name;
```

```
        $lastName = $objectRef->last_name;
        $firstInitial = substr( $firstName, 0, 1 );
        $networkLogin = strtolower( $firstInitial . $lastName );

        # assign to the custom field
        $customParams = array("entityID" => $objectId,
            "custom_$customId" => $networkLogin);
        require_once 'CRM/Core/BAO/CustomValueTable.php';
        CRM_Core_BAO_CustomValueTable::setValues( $customParams );
    }
}
```

Custom mail merge token

The CiviMail component lets you customise a bulk email message using mail merge tokens. For instance, you can begin your message with, "Hi, {recipient.first_name}!" and when John Doe receives it, he'll see, "Hi, John!" whereas when Suzy Queue receives it, she'll see, "Hi, Suzy!" You can find out more details about working with custom tokens on the CiviCRM wiki: http://wiki.civicrm.org/confluence/display/CRMDOC/Mail-merge+Tokens+for+Contact+Data.

Besides the built-in tokens, you can use a hook to create new custom tokens. Let's make a new one that will show the largest contribution each recipient has given in the past. We'll use Drupal syntax again for this one.

```
<?php

# implement the tokens hook so we can add our new token to the list of tokens
# displayed to CiviMail users
function myhooks_civicrm_tokens( &$tokens ) {
    $tokens['contribution'] =
        array('contribution.largest' => 'Largest Contribution');
    /*  just array('contribution.largest'); in 3.1 or earlier */
}

# now we'll set the value of our custom token;
# it's better in general to use the API rather than SQL queries to retrieve data,
# but in this case the MAX() function makes it very efficient to get the largest
# contribution, so let's make an exception
function myhooks_civicrm_tokenValues( &$details, &$contactIDs ) {
    # prepare the contact ID(s) for use in a database query
    if ( is_array( $contactIDs ) ) {
        $contactIDString = implode( ',', array_values( $contactIDs ) );
        $single = false;
    } else {
        $contactIDString = "( $contactIDs )";
        $single = true;
    }

    # build the database query
    $query = "
      SELECT contact_id,
         max( total_amount ) as total_amount
      FROM   civicrm_contribution
      WHERE  contact_id IN ( $contactIDString )
      AND    is_test = 0
      GROUP BY contact_id
    ";

    # run the query
    $dao = CRM_Core_DAO::executeQuery( $query );
    while ( $dao->fetch( ) ) {
        if ( $single ) {
            $value =& $details;
        } else {
            if ( ! array_key_exists( $dao->contact_id, $details ) ) {
                $details[$dao->contact_id] = array( );
            }
            $value =& $details[$dao->contact_id];
        }

        # set the token's value
        $value['contribution.largest'] = $dao->total_amount;
    }
}
```

10. DEVELOPING PAGE TEMPLATES

CiviCRM uses page template files to display all the pages. This makes it possible for a person to customize any CiviCRM screen to suit a special requirement or preference. The HTML for CiviCRM pages is not embedded in PHP code, instead a template engine (Smarty) sends the correct page to the web browser.

You should be familiar with HTML and CSS syntax to be comfortable editing page templates. Some page templates additionally make use of JavaScript and an Ajax utility, JQuery.

CHANGING PAGE TEMPLATES IS THE WRONG CHOICE WHEN ...

1. it is possible to make the needed changes by updating the CSS styles. For example, if a requirement is to hide or move some information or form fields on a screen, a CSS style for that HTML element can be changed to display: none, or position: absolute within the CSS file.
2. there is a CiviCRM hook available that can control the page. For example, there is a hook that can modify the information on the Contact Summary screen
3. there is no process in place to update the page templates after an upgrade to a new version of CiviCRM. Page templates are stored in a separate folder and are not touched during an upgrade, However new versions of CiviCRM often change which placeholder elements are available in various templates. Proper source control procedures are needed to simplify upgrades to new versions.

SMARTY TEMPLATES INTRODUCTION

CiviCRM uses a page template engine called Smarty. This documentation is focused on how Smarty is used within the CiviCRM environment. Every Smarty element is enclosed between braces like these: "{}". All the other text is going to be displayed directly as HTML in the rendered page.

Each page template is stored in a file with the extension *.tpl*. The PHP code assigns variables for content that needs to be displayed, and then lets the template engine take care of presenting it.

The Smarty template engine always does this process :

1. Load the contents of a *.tpl* file.
2. Scan the *.tpl* file for placeholder elements.
3. Replace each placeholder element with the corresponding variable value.
4. Send the resulting HTML to the web browser.

These are the most commonly used Smarty template elements:

- {$Name}: To display the value of a variable named "Name"
- {$row.Name}: To display the value of the attribute Name in the object Row
- {foreach from=$rows item=row}...{/foreach}: To loop though all the items of the Rows array
- {literal} JavaScript code{/literal} to indicate to Smarty the "{}" aren't smarty elements but JavaScript code, enclose JavaScript between {literal}
- {ldelim} and {rdelim} are alternative ways to generate { and }. This is often useful if you have a simple JavaScript code that needs a lot of values from Smarty variables
- {include file="CRM/path/to/template.tpl" param1=xxx}: includes the result of the *template.tpl*. Some included files expect to have extra param (e.g., *param1*).

Please read the Smarty documentation for more information.

Tip: To see what variables have been assigned to the template, enable debug (Administer -> Configure -> Global Settings -> Debugging) and on any URL, add **&smartyDebug=1**. It opens a new browser window listing all the variables and values.

CiviCRM introduces some extra features to Smarty:

- {ts}Any text{/ts}: It will display the translated text (if you don't use US English)
- {crmURL p='civicrm/contact/view' q="reset=1&cid=`$row.source_contact_id`"}. Generates the proper CiviCRM URL that works both on Joomla! and Drupal.
- {crmAPI} Allows retrieval and display of extra data that is not assigned to the template already. Read about the CiviCRM API for more information.

HOW TO FIND AND MODIFY THE TEMPLATES?

All the templates are under the folder *templates/CRM* in your CiviCRM installation. Finding which template is used on a given page can be difficult, but the easiest way to find out the answer is to view the source of the page from a web browser and search for ".tpl". For example, for the Contact Summary page, use the web browser to open the Contact Summary page, then click "View Source" in the browser. You should find an HTML comment such as:

```
<!-- .tpl file invoked: CRM/Contact/Page/View/Summary.tpl.
      Call via form.tpl if we have a form in the page. -->
```

You can then view the file at *templates/CRM/Contact/Page/View/Summary.tpl* to see how the HTML is generated. If you want to modify the layout; for instance to reorder the content, do **not** modify directly these files, as all the modifications will be lost on the next upgrade of CiviCRM. The proper way is to create a new folder outside of your CiviCRM folder, then navigate to "Administer -> Configure -> Global settings -> Directories" in the navigation menu, and set the complete path of the folder that is going to contain your custom templates in the field *Custom Templates*.

SCENARIO: THE CONTACT SUMMARY SCREEN NEEDS TO BE CHANGED

If you want to alter the Contact Summary page template for Acme organization, perform these steps:

1. Create the folder */var/www/civi.custom/acme/templates/CRM/Contact/Page/View*
2. Update the Custom Template field in the Global Settings directories to */var/www/civi.custom/acme/templates*. You can use any directory. We found it easier to put the custom templates under */var/www/civi.custom/yourorganisation*.
3. copy *templates/Contact/Page/View/Summary.tpl* from your CiviCRM install to */var/www/civi.custom/acme*

Tip: Say you want to modify the template for a specific profile form, or a specific event. Instead of copying the Form template to its default place (*templates/CRM/Profile/Form/Edit.tpl*), you can create a subfolder with the ID of the profile and put the template file you want to change in the subfolder (*templates/CRM/Profile/Form/42/Edit.tpl* to modify only the form for ProfileID 42).

You might want to modify a template that isn't directly the page you load, but added later via Ajax. For instance, perhaps you want to change all the tabs beside the Content Summary (Activities, Groups, etc.). The easiest way to do this is to install a development oriented plug-in to your web browser. If using Mozilla Firefox, the Firebug plug-in is indispensable. Open the Firebug console (or equivalent in your browser) and click the tab. You will see what URL has been loaded for the tab (e.g., for the notes tab: *http://example.org/civicrm/contact/view/note?reset=1&snippet=1&cid=1*). Open it in a new window or new tab of the web browser, and view the source. It also contains a comment identifying the template used (*CRM/Contact/Page/View/Note.tpl*).

Keep in mind that when you modify a template, you might have a template that doesn't work properly anymore after an upgrade of CiviCRM, because the layout has changed or the name of variables assigned to the template was modified. In our experience, the easiest is to use a source code management system (SCM) to keep track of the changes you have made. Before doing any modification of the template you copied, add it to your SCM, and obviously also commit the template after having modified it. That way, you can easily generate a patch of your changes, and see how to apply them to the latest version of the template.

SEMANTICALLY MEANINGFUL HTML ATTRIBUTES

To make it as easy as possible for you to style any element in the page (e.g. put a yellow background on all the contacts of the subtype "members"), or add Ajax (clicking on the status of the activity changes it to complete), we strive to have a consistent and coherent schema for class names and ids for the generated HTML. This makes it easier to isolate the elements you want to alter from a custom style or from JavaScript:

- There is a class **crm-entityName** defining the type of the entity bubbled up as high as possible in the DOM. For instance, each line on a list of activity has *<tr class="crm-activity ...">*
- There is an id **crm-entityName_entityID** allowing to find the id of the entity bubbled up. e.g., on a list of contacts, the contact number 42 has a *<tr id="crm-contact_42" ...>*
- Each field or column contains a class identifying it, e.g., **"crm-activity-subject"**
- Each field or column that contains a value with a fixed set of possible values (e.g., a Status, a Role, a Contact Type) contains a class identifying it. It doesn't contain the human readable version (that can be changed), but the id or a name that can't be modified by the end-user; such as **class="crm-activity-status-id_42"**. This is on the top of the class identifying the field name, so the complete HTML is *<td class="crm-activity-status crm-activity-status-id_42">Hitchhiked</td>*.

At the time of the writing, some of the templates don't follow these conventions. Please update them and submit a bug tracking issue with a patch if you need to use a template that isn't yet complying. For more information about submitting a bug or issue, read About the CiviCRM community.

DISPLAYING MORE CONTENT AND ADDING AJAX FEATURES

If your modifications go further than "simple" modifications of the layout, but need to display more content than the one assigned to the template by default, or to add Ajax functionality, use the CiviCRM API. Please read more information about using the CiviCRM API from Ajax to pursue this approach.

In most cases, using the CiviCRM APIs should be simple and only takes a few extra lines of modifications.

SOME USEFUL VARIABLES AND EXAMPLES

On each page template, you have extra Smarty variables populated by CiviCRM.

{$config} Contains a lot of useful information about your environment (including the URL, if it's Drupal or Joomla!, etc.)

{$session} Contains information about the user.

If you want to modify the template only for a logged-in user but leave it identical for anonymous users, do the following:

```
{if $session->get('userID') > 0}
Insert your modifications here
{/if}
```

11. THE API

CiviCRM has a stable comprehensive API (Application Programming Interface) that can be used to access much of the functionality of CiviCRM. The API is the recommended way for any external programme to interact with CiviCRM.

WHY USE THE API?

If you're not familiar with APIs, you might ask why you would use the API when you could access the core functions (e.g. the BAO files) directly?

The answer is that the API is guaranteed to remain stable between releases, and warn developers well in advance of any changes, where as BAOs offer no such guarantee.

There are two current versions of the API: version 2 and version 3. Version 2 is present to retain backwards compatibility with older code: it should not be used for new projects.

Version 3 is one of the seven wonders of CiviCRM. It is much more consistent than version 2 and has much improved test coverage: it should be used for all new projects.

LEARNING ABOUT (AND EXPERIMENTING WITH) THE API V3

The best place to learn about the API is your own ***test*** install of CiviCRM. Be wary about using the API on your live site because the the API does not ask for confirmation, and therefore you can easily make significant changes to your data.

Each install comes with two tools that are useful for this. The **API parameter list**, which shows all available entities which can be manipulated by the API and is available at http://[CIVICRM_URL]/civicrm/api/doc, and the API explorer, which is available at http://[CIVICRM_URL]/civicrm/api/explorer.

How to use the API

There are at least 5 different ways of using an API function:

1. as a php function, to run your own code on the same server as CiviCRM
2. via the AJAX interface, to be called from JavaScript code
3. via the REST* interface, can be called from another server via http calls
4. as a Smarty function to add data to templates
5. from drush on the command line for Drupal installations.

Calling the API in PHP

If you write custom code that is going to run within the same environment as CiviCRM, calling the API as a PHP function is the recommended way of modifying or fetching data. You will see examples of API calls throughout the developer documentation. For example, you could create several tags from a script rather than doing it manually from the CiviCRM admin page. The code below shows you how to do it using the API.

The recommended way to call the API in PHP is

```
require_once 'api/api.php';
$result = civicrm_api ($entity,$action,$params);
```

where $entity is the object you want to manipulate (a contact, a tag, a group, a relationship), and$action is get (to retrieve a list or a single entity), create (to create or update if you provide the id), delete, update and getFields (that provide the list of fields of this entity, including the custom fields).

So with real data:

```
require_once 'api/api.php';
$contact = civicrm_api('Contact','Get',array('first_name' => 'Michael', 'last_name' =>
'McAndrew', 'version' =>3));
```

Calling the API from smarty

CiviCRM defines a smarty function {crmAPI} which can be used to call the API from the template layer.

```
{crmAPI var="ContactS" entity="Contact" action="get" version="3" first_name="Michael"
```

```
first_name="McAndrew" }
{foreach from=$ContactS.values item=Contact}
  <li>{$Contact.example}</li>
{/foreach}
```

Calling the API via javascript

It also defines a javascript function crmAPI() which can be used to call the API from javascript.

```
 $().crmAPI ('Contact','get',{'version' :'3', 'first_name' :'Michael', 'first_name' :'McAndrew'}}
  ,{ success:function (data){
       $.each(data, function(key, value) {// do something  });
     }
 });
```

Calling the API via Ajax or REST

You can also call the API via Ajax or REST. Have a look at the api explorer for examples of how you call the API with these methods.

Calling the API from an external server via the REST API

The syntax and principle are identical to the other methods, but with one major difference: the external server needs to authenticate itself before being able to use any API functions.

To call the API with a browser, use:

```
http://www.example.org/path/to/civi/codebase/civicrm/extern/rest.php?q=civicrm/<function>
```

The first call is to authenticate the caller on the CiviCRM server:

```
https://eg.org/path/to/civi/codebase/civicrm/extern/rest.php?q=civicrm
   /login&name=user&pass=password&key=yoursitekey&json=1
```

The key is in the CIVICRM_SITE_KEY variable defined in your civicrm.settings.php file.

Note: On the first call, you might get an error message like "This user does not have a valid API key in the database, and therefore cannot authenticate through this interface". This means that you need to generate a key for the user, and add it to the api_key field in the civicrm_contact table in the database.

```
{"is_error":0,"api_key":"as-in-your-setting.civicrm.php",
   "PHPSESSID":"4984783cb5ca0d51a622ff35fb32b590",
   "key": "2e554f49c9fc5c47548da4b24da64681b77dca08"}
```

It returns a session id. You can then use any API adding the extra param PHPSESSID = the value returned by the log-in call.

For example:

```
http://www.example.org/civicrm/ajax/rest?fnName=civicrm/contact/search&json=1&key=
yoursitekey&PHPSESSID=4984783cb5ca0d51a622ff35fb32b590
```

Note: An action that modifies the database (such as creating a contact or group) should have the parameters passed as POST, not GET. The REST interface accepts both a GET and a POST, but you should follow the web standard and use POST for things that alter the database. You'll win the respect of your peers and the admiration of your friends and family.

AUTOCOMPLETE AND SEARCH CONTACTS (API V2)

Note that the instructions below are for API v2.

If you create a profile for individual contacts that contains the current employer, you might want to add an autocomplete on that field, such as you have on the normal contact edit form. When a user starts to type the name of the current employer, the system will attempt to autocomplete the name from your database of organisation contacts.

For security reasons, the Ajax interface is restricted to users who have access to CiviCRM - otherwise, it would be fairly easy for anyone to download all the contacts you have in your database. So that's the first thing we check for here:

```
{if $session->get('userID') > 0}

<script type="text/javascript" src="/civicrm/{$config-
>resourceBase}js/rest.js"></script>{literal}
<script>
jQuery(document).ready(function($){
  $('#current_employer').crmAutocomplete({params:{contact_type:'Organization'}});
});
</script>
{/literal}

{/if}
```

You might want to add additional filters. For instance in a profile "new volunteer from a member", you want to populate the list only with the organisations that belong to the group "members" (group id 42).

```
$('#current_employer').crmAutocomplete({params:{contact_type:'Organization',group:42}});
```

Extending the API

It should be noted that it's extremely easy to extend the api if you want to add some funky features: add a Magic.php file under api/v3, create a function civicrm_magic_getpony and voila, you have civicrm_api ("magic","getpony"), the smarty, the ajax, the REST, and it's also directly usable from the api explorer at /civicrm/ajax/doc

12. IMPORTING DATA

The CiviCRM import system can be used to import contacts, activities, contributions, event participants, and memberships. You can extend the import system to allow it to parse different data sources, such as XML, JSON, Excel, or OpenDocument. The import system is especially useful when you have legacy data in another system (such as a spreadsheet), or have data you need to migrate from one system to another.

The import system has a few important limitations. It is currently not very good at importing:

- Large data sets
- Data sets which contain information for more than component (such as contact data that also contains contributions)
- Data sets which should be imported into more than one group or tag (currently, you can choose only one group or tag for the entire import rather than specifying the value in a field of the imported data set).

These limitation apply to any extensions you build on the import. To overcome these limitations, one strategy would be to write a custom script using the CiviCRM API that parses the incoming data and makes the appropriate API calls to import it.

Note: If you're reading this in a printed copy, you may want to access the online version so you can copy and paste the code examples more easily. You can find the online version here: http://en.flossmanuals.net/civicrm

DATA SOURCES

The import system supports two data sources out of the box. CSV and SQL. CSV is exported by most spreadsheets, many web sites, and a lot of legacy systems. SQL is the language of relational databases.

Here's your first insider tip on the import system: *Every* import is an SQL import. The CSV import data source, for example, starts out by dumping the CSV into a temporary database table and then running an SQL import on those data. If you write a custom data source, it's important to remember this because it will save you a lot of mental energy. You don't need to worry about formatting or validating the import data at all. Just get it into a database table and let CiviCRM take it from there.

When would you want to write an import data source? It might be useful if, say, a client were migrating from another CRM to CiviCRM and wanted to import their data themselves. Another example would be to support a different import file format.

To make things easy on yourself, check for an existing PHP library that can read the format of the data you want to import. If you fine one, use that library to read the data into a database table.

DATA SOURCE API

The data source API is essentially a set of functions you should implement in a new PHP class. Many software developers refer to this as an "interface" or "protocol." The abstract class that defines this interface is located in *CRM/Import/DataSource.php*. You should also look at the *CRM/Import/DataSource/CSV.php* file, because this implements the interface for the CSV import feature.

Here are the functions you will implement and what they should do:

```
getInfo(); # should return an array with the title of your data source plugin

preProcess( &$form ); # if you need to set anything up before the form is
displayed, this iss the place to do it

buildQuickForm( &$form ); # here you should build a form snippet that asks the user
for the appropriate information to access the incoming data

postProcess( &$params, &$db ); # this is where you dump the incoming data into the
database table
```

After defining your import data source, you need a new Smarty template to create your form snippet. This is pretty specific to the type of import data source you're defining. Let's look at an example of a real custom data source.

EXAMPLE DATA SOURCE

This example of a custom import data source reads a JSON file and imports its contents as new contacts. This file should be named *JSON.php* and be added to the *CRM/Import/DataSource/* directory.

```
<?php

/* My awesome JSON importer of dubious utility and efficiency (but still awesome!)
 *
 * JSON should look like this:
 * {
 *   "contacts" : [
 *      {
 *        "first_name" : "Foo",
 *        "last_name" : "Bar",
 *        "other_field" : "baz"
 *      },
 *      {
 *        "first_name" : "Other",
 *        "last_name" : "Contact",
 *        "something_else" : "yep"
 *      }
 *   ]
 * }
 */

require_once 'CRM/Import/DataSource.php';

class CRM_Import_DataSource_JSON extends CRM_Import_DataSource
{
    function getInfo()
    {
        return array('title' => ts('JavaScript Object Notation (JSON)'));
    }

    function preProcess(&$form)
    {
        # nothing to do here, but we still define the function
    }

    function buildQuickForm(&$form)
    {
        ### In this function we're calling a lot of QuickForm functions.
        ### If you're unfamiliar with that library, it might be good
        ### to look up the documentation for it.

        # define a hidden field that tells the system which
        # data source class we're using
        $form->add( 'hidden', 'hidden_dataSource', 'CRM_Import_DataSource_JSON' );

        # grab the config object so we respect some system settings
        $config = CRM_Core_Config::singleton();

        # get the max upload file size from the config
        $uploadFileSize = $config->maxImportFileSize;
        $uploadSize = round( ( $uploadFileSize / (1024*1024) ), 2 );

        # assign the max upload file size to a template variable
        # see the template documentation for more info on this
        $form->assign( 'uploadSize', $uploadSize );

        # add the file selection field to the form
        $form->add( 'file', 'uploadFile', ts('Import Data File'),
            'size=30 maxlength=60', true );

        # now set the max file size so the form enforces it
        $form->setMaxFileSize($uploadFileSize);
        $form->addRule( 'uploadFile',
            ts('File size should be less than %1 MBytes (%2 bytes)',
            array(1 => $uploadSize, 2 => $uploadFileSize)),
            'maxfilesize', $uploadFileSize );

        # not a very smart rule, but it'll do for now
        $form->addRule( 'uploadFile', ts('Input file must be in JSON format'),
            'utf8File' );

        # make sure we end up with a file after the form posts
        $form->addRule( 'uploadFile', ts('A valid file must be uploaded.'),
            'uploadedfile' );
    }

    function postProcess(&$params, &$db)
    {
        # grab the name of the file that was uploaded
        $file = $params['uploadFile']['name'];

        # call a helper function we'll define below to parse the JSON
        $result = self::_JsonToTable( $db, $file,
            CRM_Utils_Array::value( 'import_table_name', $params ) );

        # grab the import table name (CiviCRM determines this for us)
        $table = $result['import_table_name'];

        # create a new ImportJob object for our table
        require_once 'CRM/Import/ImportJob.php';
        $importJob = new CRM_Import_ImportJob( $table );

        # the ImportJob modifies the table name a bit, so let's update it
        $this->set( 'importTableName', $importJob->getTableName() );
    }

    /**
     * We define this function just to keep things cleaner, the import data source
```

```
     * interface doesn't look for it; it's a private function. We use an
     * underscore at the beginning of the name to indicate this.
     */
    private static function _JsonToTable(&$db, $file, $table )
    {
        # read the JSON into a string variable
        $jsonString = file_get_contents($file);
        if (!$jsonString) {
            # oops, reading the file didn't work, generate an error
            CRM_Core_Error::fatal("Could not read $file");
        }

        # grab the config object again
        $config = CRM_Core_Config::singleton();

        # this is a bit presumptuous of us, but oh well
        $db->query("DROP TABLE IF EXISTS $table");

        # create the table where we'll store the incoming data
        $create = "CREATE TABLE $table LIKE civicrm_contact";
        $db->query($create);
        # drop the id column because the import system will add one
        $dropId = "ALTER TABLE $table DROP COLUMN id";
        $db->query($dropId);

        # decode the JSON and INSERT the records one by one;
        # it might be more efficient to build one big multi-insert,
        # but we'll leave that as an exercise for the reader

        ### BE CAREFUL THAT YOU DON'T RUN THIS ON A MASSIVE JSON FILE!!
        ### It creates one big object from the entire JSON file, so you'll quickly
        ### eat up every bit of memory PHP can use if you try to import a large
        ### file.

        # this requires that the JSON PECL extension is installed and
        # enabled in PHP
        $importObj = json_decode( $jsonString, true );

        # loop through each record in the JSON object, put it into an SQL query,
        # and insert it into the database
        foreach ($importObj['contacts'] as $newContact) {
            $fields = array_map( '_civicrm_mysql_real_escape_string',
                array_keys( $newContact ) );
            $sqlFields = "(" . implode( ',', $fields ) . ")";
            $values = array_map( '_civicrm_mysql_real_escape_and_quote_string',
                $newContact );
            $sqlValues = "VALUES (" . implode( ',', $values ) . ")";

            # construct the query and run it
            $sql = "INSERT IGNORE INTO $table $sqlFields $sqlValues";
            $db->query($sql);
        }

        # get the import tmp table name and return it
        $result = array( );
        $result['import_table_name'] = $table;
        return $result;
    }
}

# Another couple private helper functions we define
function _civicrm_mysql_real_escape_and_quote_string( $string ) {
    return _civicrm_mysql_real_escape_string( $string, true );
}

function _civicrm_mysql_real_escape_string( $string, $quote = false ) {
    static $dao = null;
    if ( ! $dao ) {
        $dao = new CRM_Core_DAO( );
    }
    $returnString = $quote ? "\"{$dao->escape( $string )}\"" :
        $dao->escape( $string );
    return $returnString;
}
```

And here's the Smarty template. It should be saved in *templates/CRM/Import/Form/JSON.tpl*.

```
<fieldset><legend>{ts}Upload JSON File{/ts}</legend>
  <table class="form-layout">
    <tr>
        <td class="label">{$form.uploadFile.label}</td>
        <td>{$form.uploadFile.html}<br />
            <div class="description">{ts}File format must be JavaScript
              Object Notation (JSON). File must be UTF8 encoded if it contains
              special characters (e.g. accented letters, etc.).{/ts}</div>
            {ts 1=$uploadSize}Maximum Upload File Size: %1 MB{/ts}
        </td>
    </tr>
  </table>
</fieldset>
```

SUMMARY

The CiviCRM Import system has a few limitations, but it is by far the easiest way for end users to get data from other systems into CiviCRM. When a client has ongoing data import needs and wants non-technical users to initiate and manage the imports, writing a custom data source may be a good solution.

If the import system cannot handle the type or amount of data you need to import, your options are to write a separate import program that calls the CiviCRM API, write your own improvements to the import system, or sponsor other developers to write improvements. If you write or sponsor improvements to the import system, make sure you contribute them back to the core system! The CiviCRM development team would be happy to discuss potential improvements with you. You can find them in the #civicrm IRC channel on irc.freenode.net.

EXTENSIONS

CiviCRM has an extensions framework that allows you share your custom searches, reports and payment processors with the community. Before you being work on writing an extension make sure someone else hasn't already contributed something similar by checking the list of extensions on http://directory.civicrm.org/.

HOW TO CREATE AN EXTENSION

Before you start, make sure your extensions are enabled! You need to configure a directory, where your extensions will be stored. To do that, you need to go to: "Administer -> Configure -> Global Settings -> Directories", fill in "CiviCRM Extensions Directory" field and click "Save" at the bottom of the page. It's best to choose a directory outside of $civicrm_root, to avoid potential problems during upgrade. Then follow the seven steps below.

1. Choose a unique key for your extension

Every extension has a unique name called the extension key. It's built using Java-like reverse domain naming to make it easier with identifying unique extensions and giving developers more flexibility on providing different extensions. So for example, if your website is civiconsulting.com and you're developing custom search showing only event payments, you can choose: com.civiconsulting.search.eventregistration as the key.

2. Create an extension directory

Go to your extensions directory and create the directory named after your key. This will be the place where you'll put all the extension related files. When you're done, it will also be put into the zip archive for further distribution.

3. Create the info.xml file

Now you need to create a file which will describe your extension so we know what a given extension does, who wrote it, who supports it and quite a few other things. It needs to be called info.xml. It's in XML format and it's relatively simple. Here's an example:

```
<?xml version="1.0" encoding="UTF-8" ?>
<extension key="org.civicrm.activity" type="search">
  <callback>ActivitySearch</callback>
  <name>Activity Search</name>
  <description>
   This custom search allows to search through activities
   and returns activities as results (not contacts as
   regular search).
  </description>
  <url>http://civicrm.org</url>
  <license>AGPL</license>
  <maintainer>CiviCRM Core Team &lt;noreply@civicrm.org&gt;</maintainer>
  <releaseDate>2010-09-01</releaseDate>
  <version>1.0</version>
  <compatibility>
   <ver>3.3</ver>
   <ver>3.4</ver>
  </compatibility>
  <develStage>beta</develStage>
  <comments>For support, please contact project team on the forums.
(http://forum.civicrm.org)</comments>
</extension>
```

Here's a quick description of the XML elements:

- **extension** - enclosing element: everything needs to sit inside of it. Attributes are:
  - **key**: unique name of the extension (should match the name of directory this extension resides in). We're going with Java like reverse domain naming to make it easier with identifying unique extensions and giving developers more flexibility on providing different extensions.
  - **type**: one of "search", "payment", "report" for now, meaning that this extension is - respectively - a custom search, payment processor or custom report.
- **callback** - the name of the class that should be called in order to run this extension. Following the namespacing convention in CiviCRM, it will also be the name of the php file. Above example means that the file $extension_dir/search/ActivitySearch.php should exist and contain Extension_Search_ActivitySearch class. Extension_Search part of the namespace is obligatory, the rest is under extension author's control.
- **name** - well, that one's easy. It's a name of the extension.
- **description** - easy as well.
- **url** - address to extensions website with documentation, more information, etc.
- **license** - the name of the license under given extension is offered.
- **mainainer** - self explanatory, hopefully. Email address inside required.
- **releaseDate** - date when given extension has been released on.
- **version** - version of the extension.
- **develStage** - if you want to push out you extension in unstable state, here's the opportunity to be sincere about it. Allowed values are: "alpha", "beta", "stable". Beta is the default if you don't define develStage - 'cause you must have been unsure, if you forgot to provide such crucial piece of information. :-)
- **compatibility**, **ver** - lists compatible versions. It cares only about first two elements of version name and ignores the rest to avoid the need for frequent updating. So if your extension is compatible with CiviCRM 3.3, also means it supports 3.3.beta4. However, if you state it supports 3.3.beta4 (*<compatibility><ver>3.3.beta4</ver></compatibility>*), it won't support any other version - no go on 3.3.1.
- **comments** - last one, the opportunity to say anything you want. :-)

Since we love things to be localised, we've added that too and you can provide the extension description in other languages - the optional **localisedInfo** section of the info file does that:

```
<extension ...>
    ...regular stuff
    <localisedInfo>
        <pl_PL>
            <description>Opis po polsku.</description>
            <maintainer>Zespół CiviCRM &lt;noreply@civicrm.org&gt;</maintainer>
            <comments>Wsparcie dla tego rozszerzenia dostępne na forum CiviCRM
(http://forum.civicrm.org).</comments>
        </pl_PL>
        <pt_BR>
            ...
        </pt_BR>
    </localisedInfo>
</extension>
```

Please note though, this only supports name, description, maintainer and comments sections - all other extension information defined here will be ignored.

4. Develop your extension!

Depending on the type of extension that you're building, you will need to follow different instructions. Please refer to separate chapters for:

- Creating A Custom Search Extension
- Creating A Report Template Extension
- Creating A Payment Processor Extension

5. Test your extension

Make sure your extension works as planned! Once you have the info.xml file, you will be able to turn the extension on and off in the Manage CiviCRM Extensions screen - use that to see what errors you're getting and react accordingly.

6. Package your extension

Once you're done with developing and testing, you need to put all the contents of your extensions directory into a zip file.

We'll start by creating a directory named exactly the same as the unique key that we're choosing for our extension. You need to prepare the info file as described above. Then once you have the info file, you can add the other required files and subdirectories to the extension package. This will be different for Custom Searches, Reports or Payment Processors - see each specific chapter for concrete examples.

7. Submit your extension for public distribution

We are working on the extension submission system, but in the meantime, just get in touch with us via info@civicrm.org with your extension or on the extensions board of the developer forum. We'll then do some testing and if everything works fine, we'll put your extension into the public distribution. Once there, everyone will be able to install it once they get to "Manage CiviCRM Extensions" screen.

14. DEVELOPING CUSTOM SEARCHES

A custom search is a method of providing new functionality within the standard CiviCRM navigation structure. This chapter looks at how to develop a Custom Search.

Custom searches produce a screen showing a set of contacts, from where you can execute actions such as sending email, printing mailing labels, and all the other actions that are available for contact search results within CiviCRM. The results page displays the same as any other search results page, such as results delivered from an Advanced Search, however, a predefined set of functions controls which information is delivered to the result page.

Custom Searches follow the Hollywood principle, "Don't call me, I'll call you." In this case CiviCRM calls your functions at the appropriate time.

CustomSearches_AggregateTotals

CustomSearches_Results

WHEN TO USE CUSTOM SEARCHES

A custom search is the right choice when ...

- you need to access the Actions list after running your search, to send email, add the contacts to a group or other actions (since the list of actions and list of tokens can be extended with developer-created hooks, the combination of a custom search + custom action + custom token is a powerful tool for implementing a special requirement. If you are interested in creating new actions or tokens, read the section on CiviCRM hooks in the chapter Extending CiviCRM.)
- you want to create a Smart Group based on the parameters of the custom search
- you want to use the search results to drive a mass mailing in CiviMail
- you want results that can be sorted by clicking any column heading in the results page.

A custom search is the wrong choice when ...

- you can use the Advanced Search or the Search Builder to create the same results
- the information you need is not primarily accessible from the CiviCRM database tables (for example, the information needs to be retrieved from a third-party system or a different database)
- the information you need does not include the CiviCRM contact ID (for example, information related to summarised event income).

CiviReport is a better choice if ...

- you need to schedule and send the entire result as an email to one or more people
- you need a summary/detail drill-down style of results page where the detail is not just the contact summary screen
- you want to use the results as a dashlet on your CiviCRM dashboard
- you need multiple report break areas within the results page, such as subtotals after every 5 records as well as grand totals
- you need to include information not related to contacts.

GETTING STARTED CREATING A NEW CUSTOM SEARCH

In this section we will create a new custom search called "BirthdaySearch" that will find all contacts whose birthdays fall in June.

Custom searches are written using PHP and SQL. Very little knowledge of PHP is needed, as you start with a template file and only make minor changes. In many cases the only changes are the SQL Select statement and which columns to display in the results.

Plan and test

Before writing the code, it is important to plan and test the SQL query and verify the results. It is valuable at this stage to review the database tables and test the SQL select statements within the database using an SQL tool such as PHPMyAdmin.

It may be helpful for you to review the information at:

- Useful CiviCRM SQL Queries: http://wiki.civicrm.org/confluence/display/CRMDOC/Useful+SQL+queries
- CiviCRM Data Architecture: http://wiki.civicrm.org/confluence/display/CRMDOC/CiviCRM+data+architecture
- CiviCRM wiki: http://wiki.civicrm.org/confluence/display/CRM/CiviCRM+Wiki

Important Note

It is important to include the contact_id field from the table civicrm_contact in your select statement. Even if you do intend to display the contact_id field, include it. The reason is that the view and edit links need the value for the cid attribute of the url. Without a value, you can not edit or view information about the contact. Follow the select statement below for guidance.

Setting the filepath

Your custom search files can be stored almost anywhere, but you must tell CiviCRM where these files are.

1. Within CiviCRM, go to: Administer > Configure > Global Settings > Directories
2. Fill in the Custom PHP Path Directory. This needs to be an absolute path to your PHP directory, such as /home2/jsmith/public_html/civicrm_custom_code/
3. Create a copy of an existing custom search file, such as the EventAggregate.php file. You will find this file at:
   <joomla root>/administrator/components/com_civicrm/civicrm/CRM/Contact/Form/Search/Custom

   or
   <drupal root>/sites/all/modules/civicrm/CRM/Contact/Form/Search/Custom
4. Place the EventAggregate.php file in the directory: /home2/jsmith/public_html/civicrm_custom_code/CRM/Contact/Form/Search/Custom
5. Rename the copied EventAggregate.php file to BirthdaySearch.php

UNDERSTANDING AND UPDATING THE SEARCH CODE

Start by opening the file BirthdaySearch.php in a text editor. The first change needed is to change the class declaration to:

```
class CRM_Contact_Form_Search_Custom_UpcomingBirthdays
   implements CRM_Contact_Form_Search_Interface {
```

Functions you will probably need to modify:

- **function_construct:** this controls the columns that are part of the results page.
- **function all:** this function is responsible for returning the SQL select statement to execute that gets the entire set of information. The select statement MUST include a field named "contact_id". Normally it also contains fields named "name" and "sort_name". For example:

```
if ( $onlyIDs ) {
     $select  = "DISTINCT civicrm_contact.id as contact_id,
      civicrm_contact.display_name as name";
    } else {
    $select = "DISTINCT civicrm_contact.id as contact_id, CONCAT(
     monthname(civicrm_contact.birth_date) , ' ', day(civicrm_contact.birth_date))
     as sort_name , civicrm_contact.display_name as name, 'birthday' as oc_type" ;

    }

    $from  = $this->from( );
    $where = $this->where( $includeContactIDs ) ;
    $sql = "SELECT $select FROM  $from WHERE $where ";

⋮
```

- **function from:** this function is responsible for returning the from clause of the SQL select statement. It is normally called from the "all" function as well as by CiviCRM.
- **function where:** this function is responsible for returning the where clause of the SQL select statement.
- **function buildForm**: this controls the results title as well as the parameters that are available to the person running the search. For this birthday search, you may want the user to be able to choose the month.

Functions you may need to modify:

- **function summary:** this function is needed if some or all columns have summary information, such as total number of birthdays in June.
- **function alterRow:** this function allows you to alter the contents of a piece of information before the results are displayed.
- **templateFile**: this function returns the name of the Smarty template that CiviCRM will use to present the results. For most purposes, the template CRM/Contact/Form/Search/Custom/Sample.tpl will suffice.

PREPARE TO RUN THE CUSTOM SEARCH

Before you can run the custom search, CiviCRM needs to be informed that it exists. This is accomplished by the following steps:

1. Go to: Administer > Customize > Manage Custom Searches.
2. Scroll to the bottom of the page and click the button New Custom Search.
3. Provide the class path as: CRM_Contact_Form_Search_Custom_BirthdaySearch
4. Provide the title as: Birthday Search

The new Birthday Search should now appear in (and can be run from) the list of custom searches in the navigation menu. It will also appear on the page reached by going to Search > Custom Searches.

The new custom search will not appear in the black navigation menu unless the navigation menu is edited. This can be done by going to Administer > Customize > Navigation Menu.

TESTING THE CUSTOM SEARCH

Always test the following behaviors of the new search:

- Test for a variety of form values, especially for invalid data. Errors in validation can lead to serious security breaches. Just because there is a drop-down list of valid months, do not assume that only valid months are passed to your custom search. Also test for values with apostrophes and other special characters.
- Test the previous and next page links for a variety of different size results.
- Test the first/last page links for a variety of different size results.
- Make a Smart Group from it and send a CiviMailing to that group.
- Test other actions in the Action menu such as Send an Email, or create PDF letters with mail merge tokens.

AN EXAMPLE OF CREATING A CUSTOM SEARCH EXTENSION

Once you've created your custom search, you can start packaging it. Let's say you will be doing an activity search.You need to prepare the info file as described in the Extensions Framework Chapter.

Sample info.xml file

```
<?xml version="1.0" encoding="UTF-8" ?>
<extension key="org.civicrm.activity" type="search">
  <callback>ActivitySearch</callback>
  <name>Activity Search</name>
  <description>
   This custom search allows to search through activities
   and returns activities as results (not contacts as
   regular search).
  </description>
  <url>http://civicrm.org</url>
  <license>AGPL</license>
  <maintainer>CiviCRM Core Team &lt;noreply@civicrm.org&gt;</maintainer>
  <releaseDate>2010-09-01</releaseDate>
  <version>1.0</version>
  <compatibility>
   <ver>3.3</ver>
   <ver>3.4</ver>
  </compatibility>
  <develStage>beta</develStage>
  <comments>For support, please contact project team on the forums.
(http://forum.civicrm.org)</comments>
</extension>
```

Then once you have the info file, you can start putting the extension package together. We'll choose "org.civicrm.activity" to be the unique identifier of the extension, so we need to give the same name to the directory that will contain our extension. Once you've created that, put the info.xml file in it.

Remember the "callback" section in info file? We've put the value "ActivitySearch" in there. Now it's time to prepare our custom search PHP class to fit in the package. First of all, put it in the file named exactly after the callback value minus the file extension part. So now we have a second file in our extension directory: ActivitySearch.php. The name of the class that is inside this file should be constructed the following way: **"Extension_<Type>_<key>_<callback>"** - which means in our case it will be: Extension_Search_org_civicrm_activity_ActivitySearch. It's rather long, but we want to avoid any problems with class name duplication. Please also note, that the extension type is capitalised, but extension key is not.

Ok, we've got the info file, there is a custom search class ready, so the last thing is a template. Just create a subdirectory of org.civicrm.activity named **templates** and put your template file there. You should name the template the same as your PHP file - so it should be ActivitySearch.tpl.

You should end up with the following structure:

```
org.civicrm.activity/
|-- ActivitySearch.php
|-- README.txt
|-- templates
|   `-- ActivitySearch.tpl
`-- info.xml
```

Let's come back to the PHP class for a minute. There are two small

15. WRITING CUSTOM REPORTS

This chapter is a brief introduction to creating a new report in the CiviReport system. You can use the same programming techniques to extend an existing report template. These tasks call for a strong grasp of PHP. More details are available in the wiki page on customizing reports: http://wiki.civicrm.org/confluence/display/CRMDOC/CiviReport+structure+and+customization.

CREATING OR CHANGING A REPORT TEMPLATE IS THE WRONG CHOICE WHEN ...

If you want a batch action on an existing result set. CiviReport does not support most of the batch actions. If your main purpose is to print or send an email to a list of contacts that meet specific criteria, write a custom search.

REPORT SPECIFICATION

Before starting on a custom report, fill out the form on the CiviCRM wiki for report specifications. Add your specification there and ask for comments in the forum or on IRC.

CREATING A CUSTOM TEMPLATE

This section creates a custom report listing all contacts' Display Name, First Name and Last Name. Note that we'll be talking about both Smarty templates (*.tpl* files) and PHP report templates--be careful not to confuse the two.
Replacing an existing report with your own version is not recommended because you may need the original too, and because an upgrade of CiviCRM will overwrite your changes with the CiviCRM version. In this section, therefore, we'll create a new Smarty template and a new PHP template for the report. You can also copy a template and report to a new location and edit them to create your custom report.

All reports are located under *CRM/Report/Form/* and grouped by component.

Create a new Smarty template and a new PHP template for the report. Because this example creates a report about contacts, we'll create a new report template named *Contact.php* and a Smarty template named *Contact.tpl*, both in a custom directory (the *Contact.php* in the custom directory for php scripts, the *Contact.tpl* in the custom directory for templates). The paths will look like:

```
PHP_CUSTOM_PATH/CRM/Report/Form/Contact/Contact.php

TPL_CUSTOM_PATH/CRM/Report/Form/Contact/Contact.tpl
```

Add a base class named **CRM_Report_Form_Contact_Contact** in *Contact.php*, Your class must inherit the form class used for the report framework. So *Contact.php* looks like:

```
<?php
require_once 'CRM/Report/Form.php';
class CRM_Report_Form_Contact_Contact extends CRM_Report_Form {
}
```

Your Smarty template file must include the report framework template. Thus, *Contact.tpl* is:

```
{include file="CRM/Report/Form.tpl"}
```

Register your report template in CiviCRM. Go to: Administer -> CiviReport -> Register Report. Enter the details shown in the following screenshot.

Reports_register

In a constructor, create an array to hold a contact, which in turn hold an array specifying two fields you want to display.

```
function __construct( ) {
    $this->_columns = array( 'civicrm_contact' =>
    array(
        'dao' => 'CRM_Contact_DAO_Contact',
        'fields' => array( 'first_name' => array( 'title' => ts( 'First Name' ) ), 'last_name'
=> array( 'title' => ts( 'Last Name' ) ),
    ) ) );
parent::__construct( );
}
```

Build the display by issuing a SELECT against the database. You can create column headers and fill in each field in the display with the results returned by the SELECT.

```
    function preProcess( ) {
        parent::preProcess( );
```

```
    }
// build select query based on display columns selected
    function select( ) {
        $select = $this->_columnHeaders = array( );
        foreach ( $this->_columns as $tableName => $table ) {
            if ( array_key_exists('fields', $table) ) {
                foreach ( $table['fields'] as $fieldName => $field ) {
                    if ( CRM_Utils_Array::value( 'required', $field ) ||
                    CRM_Utils_Array::value( $fieldName, $this->_params['fields'] )) {
$select[] = "{$field['dbAlias']} as {$tableName}_{$fieldName}";           // initializing
columns as well
           $this->_columnHeaders["{$tableName}_{$fieldName}"]['type']  =
               CRM_Utils_Array::value( 'type', $field );
           $this->_columnHeaders["{$tableName}_{$fieldName}"]['title'] = $field['title'];
                    }
                }
            }
        }
        $this->_select = "SELECT " . implode( ', ', $select ) . " ";
    }
    function from( ) {
        $this->_from = "FROM civicrm_contact {$this->_aliases['civicrm_contact']}";
    }
```

Your custom report template is ready. Press the Preview button to see the results.

After you've installed and tested your report, add it to the CiviCRM wiki as a patch to the project so that it can be used by others in the community.

AN EXAMPLE OF CREATING A CUSTOM REPORT EXTENSION

Custom reports are made of a PHP class and the template. Once we have that, we can go ahead and package it.

You need to prepare the info file as described in the Extensions Framework Chapter.

Sample info.xml file for custom report

```
<?xml version="1.0" encoding="UTF-8" ?>
<extension key="org.civicrm.report.grant" type="report">
  <callback>Grant</callback>
  <name>Grant Report</name>
  <description>Grant Report allows you to see the summary of grants that
  have been admitted to your consitutents by your organisation.  </description>
  <url>http://civicrm.org</url>
  <license>AGPL</license>
  <maintainer>CiviCRM Core Team &lt;noreply@civicrm.org&gt;</maintainer>
  <releaseDate>2010-09-01</releaseDate>
  <version>1.0</version>
  <develStage>stable</develStage>
  <compatibility><ver>3.3</ver></compatibility>
  <comments>For support, please contact project team on the forums.
(http://forum.civicrm.org)</comments>
</extension>
```

⁞Then once you have the info file, you can start putting the extension package together. We'll choose " org.civicrm.report.grant " to be the unique identifier of the extension, so we need to give the same name to the directory that will contain our extension. Once you've created that, put the info.xml file in it.

- Create the directory org.civicrm.report.grant and add the info file
- Put the report PHP class file in the directory. Remember to name it the same way as the callback attribute in your info.xml, also don't forget to follow the convention in class naming ("Extension_<Type>_<key>_<callback>" - Extension_Report_org_civicrm_report_grant_Grant in this case).
- Create org.civicrm.report.grant/templates directory and put your report's template file there

Report needs some additional information compared with custom search and we need to define it somewhere. Therefore, we'll create another XML file called report.xml. That will simply provide the extension framework with all the necessary information needed for adding it to the system. Here's how it looks:

```
<?xml version="1.0" encoding="UTF-8" ?>
  <report_template key="org.civicrm.report.grant">
    <report_url>grant/summary</report_url>
    <component>CiviGrant</component>
  </report_template>
```

Attributes explained:

We should have following directory structure containing following files:

```
org.civicrm.report.grant/
|-- Grant.php
|-- info.xml
|-- report_template.xml
`-- templates
    `-- Grant.tpl
```

Looks similar to what you have on your sandbox? Then the last thing required is to put it in the archive - "zip" it and call the file after the key defined at the beginning: org.civicrm.report.grant.zip. Congratulations, you have now packaged your first custom report.

16. PAYMENT PROCESSOR PLUGINS

CiviCRM has a plugin architecture for integration of payment processors/gateways. The core team has built plugins for PayPal Pro, PayPal Express, PayPal Web Payments Standard with IPN, and Google Checkout. JMA Consulting (Joe Murray) sponsored Alan Dixon to build and contribute a plugin for Moneris. Marshall Newrock for Ideal Solutions contributed a plugin for Authorize.net. Eileen and Lucas collaborated to contribute a plugin for DPS PaymentExpress and also wrote this document. Eileen wrote the Payment processors for FirstData, Elavon and PayflowPro - which are in the core as of v 3.1. Peter Barwell wrote eWay. Circle Interactive wrote the processors for WorldPay and SagePay.

HOW TO WRITE A PAYMENT PROCESSOR

Understand the processor you are setting up

You need to check your processor's documentation and understand the flow of the processor's process and compare it to existing processors.

Factors you should take note of are:

- Does your processor require the user to interact directly with forms/pages on their servers (like PayPal Express), or are the data fields collected by the CiviContrib form and submitted "behind the scenes"?
- What fields are required?
- What authentication method(s) are used between Civi and the payment processor servers?
- What format is data submitted in(soap/xml, arrays...)?
- What are the steps in completing a transaction (e.g. simple POST and response, vs multi-step sequence...). Are transaction results returned real-time (PayPal Pro/Express and I think Moneris) - or posted back in a separate process (PayPal Standard w/ IPN)?

Note that none of the plugins distributed with CiviCRM use a model where the donor's credit card info is stored in the CiviCRM site's database. For PayPal Pro, Authorize.net and PayJunction - Credit card numbers, exp date and security codes are entered on the CiviCRM contribution page and immediately passed to the processor / not saved. For PayPal Std, Google Checkout - the info is entered at the processors' site.

Determine what 'type' of processor you are dealing with.

Figuring this out will set you on the right track. In CiviCRM there are three 'billing types':

form - a form is where the credit card information is collected on a form on your site and submitted to the payment processor

button - buttons rely on important information (success, variables etc) being communicated directly between your server and the payment processor. (E.g. in the paypal express method, the customer is transferred to the server to enter their details but the transaction is not pushed through until an html request is sent from your server to the processor and the server replies with the response. The server also uses html to query certain variables from the server. CURL is used for this.) The user's session remains intact but I'm not sure if session variables or variables sent from the payment processor are used to identify the transaction and customise what the user sees

notify - the notify method deals with a situation where there is not a direct two way communication between your server and the processor and there is a need for your server to identify which transaction is being responded to. Some processors may have more than one confirmation: e.g. one html GET via the user's browser and a later html GET from the payment processor server. If the user's browser never returns the processor needs to be able to figure out which transaction is involved & to complete it. If the GET is from the user's browser it needs to do the same thing but also redirect the user appropriately.

Add the processor into the processor_types table

Add a line to `civicrm_payment_processor_type` - eg. using php myadmin or mysql

Fields are:

```
ID   - unique ID
DomainID - generally 1
Name - this name needs to be used as the name for the various php files as well.
Description (optional field)
Isactive  - boolean
```

```
Isdefault - boolean
user_name_label - this is what the username field is described as in the configure payment
instruments screen
password_label - likewise
signature_label
class_name - name of class (in code) should be Payment_xxx where xxx is the same as the name
field
various self-explanatory url fields & then
BillingID - number from 1-4 - this is the important one - will add more later
Isrecurr - boolean
```

The BillingID will determine the process followed. ID types are:

1 = form (onsite processing with SSL set up)

3 = button (e.g. PayPal Express)

4 = notify (redirect to offsite processing)

Store any function files from your payment processor

Create an appropriately named folder in the 'packages' directory for any files provided by your payment processor which have functions you need.

Create files in the components for your processor

There needs to be a file for your processor in CRM\Event\Payment and in CRM\Contribute\Payment. The name needs to be the same as the name you inserted into the processor_types table. These are pretty easy to copy from others as they don't have much in them.

Write your processor

OK, the groundwork is laid but writing the processor is the hard bit.

Depending on your billing mode there are different considerations - I have less information and it is less checked on the first two. The file will live in CRM\Core\Payment and have the same name as entered into your processor_type table.

Test your processor

Some suggestions of what you might test are here

http://wiki.civicrm.org/confluence/display/CRMDOC/Testing+Processor+Plugins

EXAMPLES OF PROCESSOR CODE

Form Mode

The function called by this billing mode is

```
doDirectPayment()
```

If you return to the calling class at the end of the function the contribution will be confirmed. Values from the $params array will be updated based on what you return. If the transaction does not succeed you should return an error to avoid confirming the transaction.

The params available to doDirectPayment() are: -

```
qfKey -
email-(bltID) -
billing_first_name (=first_name)
billing_middle_name (=middle_name)
- billing_last_name (=last_name)
- location_name-(bltID) = billing_first_name + billing_middle_name + billing_last_name
- streeet_address
-(bltID)
- city-(bltID)
- state_province_id-(bltID) (int)
- state_province-(bltID) (XX)
- postal_code-(bltID)
- country_id-(bltID) (int)
- country-(bltID) (XX)
- credit_card_number
- cvv2 - credit_card_exp_date - M - Y
- credit_card_type
- amount
- amount_other
- year (credit_card_exp_date => Y)
- month (credit_card_exp_date => M)
- ip_address
- amount_level
- currencyID (XXX)
- payment_action
- invoiceID (hex number. hash?)
```

bltID = Billing Location Type ID. This is not actually seen by the payment class.

Button Mode

the function called by this billing mode is

```
setExpressCheckout()
```

The customer is returned to confirm.php with the rfp value set to 1 and

```
getExpressCheckoutDetails()
```

is called when the form is processed

```
doExpressCheckout()
```

is called to finalise the payment - a result is returned to the CiviCRM site.

NOTIFY MODE

The function called is

```
doTransferCheckout()
```

The details from here are processor specific but you want to pass enough details back to your function to identify the transaction. You should be aiming to have these variables to passthrough the processor to the confirmation routine:

```
contactID
contributionID
contributionTypeID
invoiceID
membershipID(contribution only)
participantID (event only)
eventID (event only)
component (event or contribute)
qfkey
```

Handling the return can be the tricky part.

In order to keep the return url short (because paymentexpress appends a long hex string) our return url goes to a file (in the extern folder) which redirects through to the 'main' routine in paymentExpressIPN.php (IPN = instant payment notification).

note - you need to re-initialise the environment to get civi functions to work

```
require_once '../civicrm.config.php';
require_once 'CRM/Core/Config.php';
$config =& CRM_Core_Config::singleton();
 An appropriate structure for the return routine file is:
function newOrderNotify( $success, $privateData, $component,$amount,$transactionReference ) {
        $ids = $input = $params = array( );
}
```

this version in the paymentexpress file is not processor specific - pass it the variables above and it will complete the transaction. Success is boolean, the private data array holds, the component is (lower case) 'event' or 'contribute' , amount is obvious, transaction reference is any processor related reference.

contactID, contributionID, contributionTypeID,invoiceID, membershipID(contribution only), participantID (event only), eventID (event only)

static function getContext($privateData, $orderNo)

generic function - taken from google version - retrieves information to complete transaction (required?)

private data as above

orderno - transactionreference is OK

Static function main (blah, blah, blah)

this function is processor specific - it converts whatever form your processor response is into the variables required for the above function and if necessary redirects the browser using

CRM_Utils_System::redirect($finalURL);

TESTING PAYMENT PROCESSOR PLUGINS

Here's some suggestions of what you might test once you have written your payment processor plug in.

Don't forget that you need to search specifically for TEST transactions

i.e. from this page /civicrm/contribute/search&reset=1 chose 'find test transactions'

Std Payment processor tests

1) Can process Successful transaction from

- Event
- Contribute Form
- Individual Contact Record

Transaction should show as confirmed in CiviCRM and on the payment processor

2) Can include , . & = ' " in address and name fields without problems. Overlong ZIP code is handled

3) Can process a failed transaction from a Contribute form

Can fix up details & resubmit for a successful transaction

e-mail address is successfully passed through to payment processor and payment processor sends e-mails if configured to do so.

The invoice ID is processed and maintained in an adequate manner

7) Any result references and transaction codes are stored in an adequate manner

Recurring Payment Processor tests

NB - IN Paypal Manager the recurring billing profiles are in Service Settings/Recurring Billing/ Manage Profiles

1) Process a recurring contribution. Check

- wording on confirm page is acceptable
- wording on thankyou pages is acceptable
- wording on any confirmation e-mails is acceptable
- the payment processor shows the original transaction is successful
- the payment processor shows the correct date for the next transaction
- the payment processor shows the correct total number of transactions and / or the correct final date

2) Try processing different types of frequencies. Preferably test a monthly contribution on the last day of a month where there isn't a similar day in the following month (e.g. 30 January)

3) Process a transaction without filling in the total number of transactions (there should be no end date set)

4) Process a recurring contribution with the total instalments set to 1 (it should be treated as a one-off rather than a rec urring transaction). It should not show 'recurring contribution' when you search for it in CiviCRM

5) PayflowPro - check that you can edit the frequencies available on the configure contribution page form

6) Depending on your processor it may be important to identify the transactions that need to be updated or checked. You may wish to check what it being recorded in the civicrm_contribution_recur table for payment processor id, end date and next transaction date.

Specific Live tests

1) Successful and unsuccessful REAL transactions work

2) Money vests into the bank account

3) For recurring transactions wait for the first recurrent transaction to vest

AN EXAMPLE OF PACKAGING A PAYMENT PROCESSOR

Start by creating a directory named exactly the same as the unique key that you're choosing for your extension. Let's use the example of a Google Checkout payment processor - both the key and the name of directory will be org.civicrm.googlecheckout.

Sample info.xml file for a Payment Processor

```
<?xml version="1.0" encoding="UTF-8" ?>
 <extension key="org.civicrm.googlecheckout" type="payment">
  <callback>Google</callback>
  <name>Google Checkout</name>
  <description>Google Checkout Payment Processor</description>
  <url>http://civicrm.org</url>
  <license>AGPL</license>
  <maintainer>CiviCRM Core Team<noreply@civicrm.org></maintainer>
  <releaseDate>2010-09-01</releaseDate>
  <version>1.0</version>
  <develStage>stable</develStage>
  <compatibility><ver>3.3</ver></compatibility>
  <comments>For support, please contact project team on the forums.
(http://forum.civicrm.org)</comments>
</extension>
```

Then once you have the info file, you can start putting the extension package together.

- Create the directory org.civicrm.googlecheckout and add the info file
- Put the report PHP class file in the directory. Remember to name it the same way as the callback attribute in your info.xml, also don't forget to follow the convention in class naming ("Extension_<Type>_<key>_<callback>" - Extension_Report_org_civicrm_report_grant_Grant in this case).

Payment processors do not require templates; however, we need to do some more stuff in order for our extension to work. First of all, this payment processor requires some external libraries to work. Let's put them in packages directory inside org.civicrm.googlecheckout. So now, we have the following structure:

```
org.civicrm.googlecheckout/
|-- CRM
|   |-- Contribute
|   |   `-- Payment
|   |       `-- Google.php
|   `-- Event
|       `-- Payment
|           `-- Google.php
|-- Google.php
|-- info.xml
|-- packages
|   `-- Google
|       |-- README
|       |-- demo
|       |   |-- cartdemo.php
|       |   `-- responsehandlerdemo.php
|       |-- googlemessage.log
|       `-- library
|           |-- googlecart.php
|           |-- googleitem.php
|           |-- googlemerchantcalculations.php
|           |-- googleresponse.php
|           |-- googleresult.php
|           |-- googleshipping.php
|           |-- googletaxrule.php
|           |-- googletaxtable.php
|           `-- xml-processing
|               |-- xmlbuilder.php
|               `-- xmlparser.php
`-- processor-type.xml
```

Finally, there is one more file. We are going to need some additional information - hence the processor-type.xml file. It's contents reflects the contents of civicrm_payment_processor_type database table. Here's an example for Google Checkout payment processor.

```
<?xml version="1.0" encoding="UTF-8" ?>
 <payment_processor_type>
  <user_name_label>Merchant ID</user_name_label>
  <password_label>Key</password_label>
  <signature_label></signature_label>
  <subject_label></subject_label>
  <class_name>Payment_Google</class_name>
  <url_site_default>https://checkout.google.com/</url_site_default>
  <url_api_default></url_api_default>
  <url_recur_default></url_recur_default>
  <url_button_default>https://checkout.google.com/buttons/checkout.gif
?merchant_id=YOURMERCHANTIDHERE&w=160&h=43&style=white&variant=text&loc=en_US</url_button>
 <url_site_test_default>https://sandbox.google.com/checkout/</url_site_test_default>
  <url_api_test_default></url_api_test_default>
  <url_recur_test_default></url_recur_test_default>
<url_button_test_default>https://sandbox.google.com/checkout/buttons/checkout.gif
?merchant_id=YOURMERCHANTIDHERE&w=160&h=43&style=white&variant=text&loc=en_U
  <billing_mode>notify</billing_mode>
  <is_recur>0</is_recur>
  <payment_type>1</payment_type>
 </payment_processor_type>
```

As with other Extensions, the last thing required is to put it in the archive - "zip" it and call the file after the key defined at the beginning: org.civicrm.report.grant.zip. Congratulations, you have now packaged your first Payment Processor.