Developer Zone

GemsTracker is a tool for tracking fixed sequences of surveys asked to groups of individuals or organizations.

GemsTracker is not a tool for building and administrating surveys. For that specialized survey software packages are used by GemsTracker, with LimeSurvey as the first among equals.

GemsTracker is a tool that allows:

follow up surveys surveys dependent on answers in previous surveys multiple people to answer surveys concerning one person different sequences of surveys (i.e. tracks) assigned to different people For developers it offers a broad library of web pages that can be easily extended or overridden on a per project basis.

Working with the GemsTracker library can sometimes be difficult. We have a rich API documentation but sometimes you need a little bit more to get you going. This is the right place to start searching or adding your own information.

GemsTracker is developed to allow you to quickly and easily adapt it to the demands of specific projects. We quickly found out that almost every project has different data demands for respondents. I.e. some projects need a respondents postal address, others need only a birthday while requiring their fathers birthday as well. For this reason GemsTracker uses a separate project application directory that can extend, change and overrule almost any feature of the core GemsTracker libraries.

GemsTracker itself is built on other software, both open source as well as commercial. This page describes all these layers:

Application / Project layer: your own code (optional) GemsTracker Layer: GemsTracker and (optionally) LimeSurvey Programming language layer: PHP and the Zend Framework Server software layer: Apache or IIS and MySQL OS layer: Unix, Linux or Windows Server or Workstation The last three layers are described first in the Software requirements. Next comes an overview of the GemsTracker Core and lastly the How to's to get started with adapting the software for a specific project.

Software requirements

GemsTracker is built in PHP 5.3 on top of the Zend Framework and uses MySQL 5. The development team runs software runs the software on both Windows and Unix systems using both Apache 2 and IIS 7 as servers. Additional web publication environments that support PHP 5.3 and MySQL should work, but we have not tested them so you are on your own.

To use GemsTracker you need:

a working Webserver (Apache or IIS) PHP 5.3 or higher Zend Framework 1.11 or higher access to a database on a MySQL server version 5 or higher (optionally) a LimeSurvey 1.90 or higher installation GemsTracker is built with a plug-in survey engines, but the only survey engine currently available in open source is Lime Survey. At least one other survey systems has been implemented but it is not available in open source.

The GemsTracker Core

GemsTracker is a library that extends the Zend Framework libraries, but also supplies a standard new project template.

A new project

GemsTracker supplies a default new_project installation that allows you to quickly setup a standard project. The package contains these directories:

application Project specific code and settings classes Project specific classes configs Project specific Zend Framework, GemsTracker and database configuration files controllers (Optional) Project specific Zend FW controllers events (Optional) Project specific track engine events languages (Optional) Overrule the default translations layouts (Optional) Project specific Zend FW layouts snippets (Optional) Project specific code that generates Html views (Optional) Project specific Zend FW views htdocs The webroot of the project gems Default GemsTracker css, images and jQuery library The location for the GemsTracker library and optionally the Zend FrameWork? Gems The GemsTracker library directory, link to a stable release or to our development branch var A writable directory for files that can change after installation cache Cache and sessions logs Error logs settings File locks and other installation dependent settings uploads Pdf's ant other uploadable files The library

GemsTracker is implemented as a separate library. You should either link to a stable tagged release and not change the code, or if you are involved in the project you can develop against the GemsTracker trunk.

The directory structure is based on the standard Zend Framework project directory structure with some extensions.

classes The main project code configs Database definitions for project types controllers Zend FW Controllers stubs that can be overruled by a project docs Some documentation languages Default translations (currently Dutch and English) layouts GemsTracker default Zend FW layouts snippets Code that generates reuseable Html fragments / snippets views GemsTracker default Zend FW views (but very empty for a Zend project) GemsTracker builds on the Zend Framework but does not follow in slavishly. Some changes are caused by the requirement that standard code can be overruled at the project level. Other changes were made because we want to extend the Zend Framework (but did not yet get round to the extensive documentation and unit testing required by the framework). These two types of extensions are easily distinguished by their parent directories within the classes directory:

Gems The core of GemsTracker MUtil Possible extensions to Zend, should not be Gems specific ZFDebug A Google build debug extension for Zend Zend Those extreme exceptions where we really had to fix a Zend bug (currently only one) GemsEscort.php The Zend Bootstrap object for GemsTracker. This object handles initialization, layout and security. The GemsEscort? object must be overloaded at the project level. For Gems and MUtil we generated API documentation, but here we will describe their effect on Zend Framework development.

The MUtil extensions

MUtil stand for MagnaFacta? Utilities. MagnaFacta is one of the companies hired to develope GemsTracker.

When a MUtil directory has the same name as an existing Zend directory it usually concerns a simple extension of that Zend component. E.g. the Markup directory contains an extension that renders e.g. BB or Wiki code to flat text. Always useful if you want to include a text version for smartphone with an HTML e-mail message. The Potemkin Translate adapter allows you to act as if there is a translator, without defining any.

However some other default extensions are more important:

Application Extends the Zend Bootstrap object to an Escort object that allows .NET like event function use during the whole application request cycle. Controller Extends the standard Action to include the use of new Html, Models and Snippets components Form Provided extensions for using both tabbed and tabled forms, improved focus handling and use of the Html component. Other directories extend the framework. These can be divided in two sets, high-level and low-level. We start with the low level extensions. The low-level do not adapt the Zend Framework, but enable the other extensions:

Lazy Delayed execution, think callable with parameters and repetition Parser An SQL parser for SQL script processing Ra Array and parameter processing functions Registry Automated object parameter loading using a registry Util Programmer extendable functionality (e.g. for factory functions in Ra, Html and Snippets) The high level packages are the ones that make a GemsTracker a non standard Zend Application:

Html Simple extendable HTML classes using Lazy repetition and Ra parameters Model The M in Model View Controller. Describe labeling, display formatting, validating, etc... for non-db or db-based data sets Snippets Quickly create reusable HTML fragments from code. Use of Html, Model and Registry is prepared but not required The Gems extension

Most components in Gems extend a Zend or MUtil component adding functionality specific to GemsTracker, without adding any significant changes to the existing workings of those components. The exceptions fall in two broad categories: those that enable extensions and changes at the project level and those that form the core of GemsTracker.

Project level extensibility GemsTracker tries to give a programmer as much freedom as necessary to change the core workings at the project level, without the programmer having to change or copy the core GemsTracker library. Default Standard controllers, for easy overloading of controllers at the project level Loader Allow 'mirrored' project level objects to be loaded instead of Gems level objects Project Choose multi-layout, multi-organization, logging level and track types GemsTracker functionality Communication (SOAP) communication with external applications Event Survey or track specific code triggered before or after a survey is taken Export Data export for scientific analysis Menu The application menu, of course adaptable at the project level Tracker THE CORE: integration with survey sources, track engines and token display User Extensible, role based user authentication and authorization How to's

The Zend Framework may or may not be an easy framework for PHP development, but a GemsTracker project can be adapted without deep knowledge of the Framework. Of course when a project needs extensive adaptations (e.g. a separate survey engine or intensive integration with other applications) knowledge of the Zend Framework becomes necessary, but there is no need to dive in at the deep end of the framework.

Here are some examples of common extensions.

How do I get started

Follow the steps in the Quickstart guide.

How do I create my own controller

Check Zend FW Quickstart for an intro on controllers and their names and to get an introduction to the Zend naming conventions and project setup.

Most GemsTracker controllers use a Model (i.e. display and browse one set of data) and inherit from either Gems_Controller_ModelSnippetActionAbstract (the newer solution) or Gems_Controller_BrowseEditAction (old solution). Check Gems_Default_StaffAction for an example of the first and Gems_Default_GroupAction for the second type of standard controller.

In other cases, e.g. when you just want to output some (mostly) fixed HTML, you can use 'Gems_Controller_Action' as a template. Gems_Default_ContactAction is a good example.

Of course you can also use your existing Zend controller. All Zend application variables will be set

Do not mirror the directory location from the Gemstracker library. The classes/Gems/Default' location is used to ease the adaptation of existing controllers at the project level. Use insteaf the Zend Framework method of creating xxxController with same filename and object name in application/controllers`.

How do I add my own controller to the menu?

You successfully created your HelloController?, but the you called the World action and Gems tells you you are not allowed to access the page. This is because all access to pages is controlled through the Menu object. GemsTracker has a default menu, but you can change it to suite your needs.

Go to the /application/classes/[project name]/Menu.php file and create/edit the loadProjectMenu() function.

```
    public function loadProjectMenu()
    {
       // Hello world page
       $this->addPage('Hello', null, 'hello', 'world');
    }
```

And voila: 'Hello' appears as a menu choice and you can select the controller.

The definitive place to check the workings of the addPage() function is of course the API documentation (for Gems/Menu/MenuAbstract→addPage()) or to load the source in your program editor. But here is how the addPage function works.

parameters label The label for display in the menu, leave null when used, but not displayed in the menu. privilege When empty the action is always accessible, specify 'pr.islogin'/'pr.nologin' when a user must/must not be logged in. Specify your own string when you want te set the privilege yourself for specific application roles. controller The name of the new controller. action The name of the action or 'index' by default. other An optional array for advanced usage. You should really have a look at the API documentation. return: a Gems_Menu_SubMenuItem object where you can specify sub items So putting it together you can add something more complicated.

```
    public function loadProjectMenu()
    {
       // Hello world page
       // - always accessible
       // - move to top of menu using 'order'
       $page  = $this->addPage('Hello', null, 'hello', 'index', array(
           'order' => 0   // Put this page at the top of the menu
       ));
```

```
        // Add sub-page for logged in user
        $page->addPage('You', 'pr.islogin', 'hello', 'you');

        // Add sub-page for when not logged in
        $page->addPage('World', 'pr.nologin', 'hello', 'World);

        // Add sub-page for roles that you gave the my.secret privilege
        $page->addPage('Secret', 'my.secret', 'hello', 'secret');

        // Add sub-page for everybody
        $page->addPage('Everybody', null, 'hello', 'everybody');

        // Add sub-page that is not displayed, but that you can access when
you know the url
        $page->addPage(null, null, 'hello', 'hidden');
    }
```

How do I change the displayed columns in the respondent/patient overview?

As every project has it's own data to work on, one of the most common actions is to change the display of the columns shown in the Respondents/Patients screen.

First to explain why we sometimes talk about respondents and sometimes about patients. Of course a respondent is someone who either answers a survey or about whom a survey is answered by someone else. Either way the survey belongs to that respondent. A patient is of course someone who receive health care. As GemsTracker is build for health care institutions in most existing projects respondents are patients, but there is no reason why they could not be truckers, butchers, mail man or just any other group of random respondents. That is why internally GemsTracker revers to respondents (we even hunt down the use of the word patient in the core library) but use a default English translation that does not but translate 'respondent' into 'patient'.

To change the displayed columns copy library/Gems/controllers/RespondentController.php' to application/controllers/RespondentController.php'. GemsTracker automatically sees the new RespondentController.php file and starts using that controller. RespondentController.php is an empty stub that inherits all functionality from Gems_Default_RespondentAction. This allows you to overrule or extend the default functionality without have to copy the code, with all the maintenance issues this entails.

Now your first thought might be that to change the workings of the 'index' action you must overrule the indexAction() function, but the search as you type function returns a result from autofilterAction() and column content is added in the protected _createTable() function that is used by both functions. However, the actual action of specifying what columns have to be displayed is done in the addBrowseTableColumns() function overloaded in RespondentAction. To change the columns you must overload this function again in your copy of RespondentController.

Now you might have noticed that the Respondent index screen displays quite a complicated table. The data is displayed in rows, but most cells have two lines in them, e-mail addresses are links when they exists, there are some extra texts that appear only when needed. There is a paginator, you can change the number of rows, the headers are sort links. There are buttons on the sides and if you click on a row, it acts as if the first button was clicked. You noticed all that? Thank you! Did you also notice that you can use the Page Up and Page Down keys to browse, Ctrl Up and Ctrl Down to see more or

less rows, and use the normal arrow keys to select a row and then press enter to open it? Thought not. Well that all works too.

It must take hundreds of lines of code to write it, no? Well yes, it does take hundreds of lines, but if we look in RespondentAction we see the code actually written in the function is quite limited. All the magic work is done by the TableBridge and AbstractModel objects.

```
    protected function addBrowseTableColumns(MUtil_Model_TableBridge
$bridge, MUtil_Model_ModelAbstract $model)
    {
        $model->setIfExists('gr2o_opened', 'tableDisplay', 'small');
        $model->setIfExists('grs_email',   'itemDisplay',
'MUtil_Html_AElement::ifmail');

        if ($menuItem = $this->findAllowedMenuItem('show')) {
$bridge->addItemLink($menuItem->toActionLinkLower($this->getRequest(),
$bridge));
        }

        // Newline placeholder
        $br = MUtil_Html::create('br');

        // Display separator and phone sign only if phone exist.
        $phonesep = $bridge->itemIf($bridge->grs_phone_1,
MUtil_Html::raw('&#9743; '));
        $citysep  = $bridge->itemIf($bridge->grs_zipcode,
MUtil_Html::raw('  '));

        $bridge->addMultiSort('gr2o_patient_nr', $br, 'gr2o_opened');
        $bridge->addMultiSort('name',            $br, 'grs_email');
        $bridge->addMultiSort('grs_address_1',   $br, 'grs_zipcode',
$citysep, 'grs_city');
        $bridge->addMultiSort('grs_birthday',    $br, $phonesep,
'grs_phone_1');

        if ($menuItem = $this->findAllowedMenuItem('edit')) {
$bridge->addItemLink($menuItem->toActionLinkLower($this->getRequest(),
$bridge));
        }
    }
```

This function combines all the power op the MUtil Html, Lazy and Model components combined with the Gems Menu object. Thankfully you do not need to know all these objects well to work with them.

E.g. the two if () statements are just the way a menu item choice is added. Copy the code and it will work. Remove it and the buttons disappears. The buttons will also disappear when you login as a user that may not use them. Usually that is the way to go with menu items.

The model is in this case actually a Gems_Model_RespondentModel, which is a database using JoinModel that combines multiple tables. It was created by the function RespondentAction→createModel(), that uses the Gems_Model→getRespondentModel() function to

create the model. The extendability of GemsTracker is best demonstrated by the fact that you can change this model both in the addBrowseTableColumns() function and in all the other functions or create a YourProject_Model_RespondentModel and it all works.

The 'grs_' and 'gr2o_' strings you see are field names in the tables gemsrespondents and gemsrespondent2org table that are both in the respondent model. name is the name of an SQL column added to the model that displays the result of an SQL expression that just so happens to return the full name of the respondent. The variables are just constants, though with some magic Lazy functionality in the itemif() function.

The addMultiSort() functions add a cell to the $bridge (that is used to form a bridge between an HTML table and a model). Add a field name that exists in the model and it will be displayed in that column. Add a MUtil_Html::raw() object to add fixed text. Just experiment with commenting field on or off and you will quickly get the result you want.