Rapid Application Development

with QCubed

A step by step introduction guide
Updated for QCubed v21.0

Author: Gianni Rossini –Ma.Gi.A. Informatica
magiainformatica@alice.it

Copyright 2010 © QCubed Development Team, http://qcu.be

Distributed under the MIT License, http://www.opensource.org/licenses/mit-license.php

TABLE OF CONTENTS

Table of Contents 2
Introduction 7
The Code Generator 7
Event-driven, Stateful user interface Framework 7
Project Vision 7
Note on Qcodo Backward Compatibility 8
Changelog and bug fixes: 8
Chapter 1: Installation 10
1.1. Prerequisites 10
Intended audience 10
Web Server 10
PHP 5 10
Database 10
1.2. Installation 10
Windows 10
Step 1: Downloading QCubed 11
Step 2: Unpacking QCubed 11
Step 3: Moving files to their proper location 11
Step 4: Setting permissions 11
Step 5: Setting the DOCROOT 11
STEP 6: Setting Database Parameters 12
Linux 14
Step 4: Setting permissions 14
Step 5: Setting the DOCROOT_SUBFOLDER and database parameters 14
Chapter 2: Introduction to QForms 15
2.1. Definition 15
2.2. Our first QForms application 15
Example breakdown 16

Adding Controls and State 17
Adding events 18
Summary 20
Chapter 3: Introduction to Code Generation 21
What QCubed does for you 21
Code Generation of Data Objects 21
HTML Form Drafts 21
What you still have to do 21
Example 21
Database layout 21
Code generation 22
Overview of the generated files 23
Viewing the result 24
Chapter 4: Understanding the generated code 25
Data Objects 25
QForm and QPanel - Drafts and Dashboard 26
Meta Controls 26
meta Controls DataGrid 27
Summary 27
Chapter 5: More on code generation 28
Creating the database 28
Foreign keys 28
MyISAM and foreign keys 29
Code generation – step 1 29
Adding data 30
Overriding the default return string for an object 32
Extending the database 33
Code generation – Step 2 33
Look and feel personalization. 34

Adding time tracks 34
Changing default values 35
Reuse available code 35
To use Form 35
To use Panel 37
Now we can return to intervention required to change default value. 41
Adding validation in Forms 43
Adding validation in Panels 45
Summary 46
Chapter 6: Putting it together: Creating a QCubed application 50
Changing the database model 50
Adding encryption and validation of the password 50
Adding an extra field 51
Validation 53
Password box – how we hid the password 54
Encryption 54
Chapter 07: Creating the application 57
Step 1: Create the initial application page – index.php 57
header.inc.php 57
menu.inc.php 57
footer.inc.php 58
index.php 58
Step 2: creating a login form 59
login.php 59
login.tpl.php 60
Step 3: verifying the user 61
Step 4: Creating a protected page 63
protected.inc.php 63
Step 5: Logout 64

Step 6 :Add time page 65
Go magically to Timetrack panel(excerpt from Chapter 05) 65
index.php (magic generic code located in table related directory) 65
Step 7: Modifying timetrack Panel Edit 69
timetrack/panel_drafts.php 69
TimetrackEditPanel.tpl.php 69
TimetrackMetaControl.class.php (**in includes/meta_controls**) 70
Step 8: change password page 71
changepass.php 71
changepass.tpl.php 72
changepass.php 72
Step 9: Manage users and projects 73
menu.inc 74
Summary 75
Chapter 8: QCubed Actions and Events 76
Basic Qform - a look over QForms and Qcontrols again 78
Understanding State 78
Understanding the QForm Process Flow 79
The Four-Function Calculator: Our First Simple Application 80
Learning about Validation 81
Custom Renderers and Control Properties 81
Basic AJAX in QForms - a look at how to AJAX-enable your QForms 82
More About Events and Actions – 83
Combining Multiple Actions on Events 83
Making Events Conditional 83
Triggering Events after a Delay 84
Triggering Arbitrary JavaScript, Alerts and Confirms 84
Other Client-Side Action Types 85
Paginated Controls - The QDataGrid and QDataRepeater controls 85

The Examples Site Database 85
Paginated Controls - The QDataGrid and QDataRepeater controls 89
An Introduction to the QDataGrid Class 89
The QDataGrid Variables -- $_ITEM, $_COLUMN, $_CONTROL and $_FORM 89
Sorting a QDataGrid by Columns 90
Adding Pagination to Your QDataGrid 90
Enabling AJAX-based Sorting and Pagination 90
Simple ergonomic tip on datagrid interface 91
Advanced Controls Manipulation - 94
Let our timetrack code survive on new db created for examples 94
Login.php modification 94
Add application directory (related to new table) 98
One artist's touch on login screen 100
Summary 102
Chapter 9: QCubed Objects reference guide 103
General Package 103
Interfaces 103
Classes 103
Functions 104
Files 105

INTRODUCTION

QCubed (pronounced "Kju'– cubed") is a PHP5 Model-View-Controller framework. The goal of the framework is to save the time for developers around mundane, repeatable tasks, allowing them to concentrate on things that are useful AND fun.

How many times have you written that SQL query, and then parsed out the results? How about that time when you had to create a form with validation logic? How about a situation where you had to move your database back-end from MySQL to PostgreSQL or another database?

All of these situations, and many more, can be simplified with QCubed. There are two key elements to the framework: the Code Generator, and the event-driven, stateful user interface framework (QForms).

THE CODE GENERATOR

The Code Generator creates PHP classes based on your database schema. It uses the concept of ORM, object-relational mapping[1], to map your DB tables to PHP classes, to allow you to manipulate objects, instead of constantly issuing SQL queries. One-to-many relationship? No problem. Association tables? No problem. Ease of transitioning between RDBMS systems? That's the whole point. Object-oriented querying? We got it.

EVENT-DRIVEN, STATEFUL USER INTERFACE FRAMEWORK

QCubed comes with a built-in user interface library, called QForms. QForms provide a framework for a true model-view-controller infrastructure in your application. Using standard HTML, create a layout of your page (view). Insert a few controls into that HTML to make it a template that will display the form data. Define those controls and their logic in a PHP class that derives from QForm (controller). Use the code-generated ORM classes to read and write from the database (model).

Customize and extend any component of the system: override properties of a QForm; create your own custom control; use a combination of controls to define a reusable QPanel that can be used as a building block across multiple pages. Abstract out the complex database logic into customizable ORM classes.

PROJECT VISION

QCubed framework takes roots at QCodo, an excellent framework developed by Mike Ho, which, unfortunately is managed solely by him, and when real life interferes has had gaps in updates stretching for over a year. QCubed is a community effort to take QCodo forward and to keep moving regardless of real life. We have a committed team of core contributors. We have built businesses on top of this framework; we are constantly working to make it more stable, robust, and constantly growing. We very much encourage contributions from a wider community

[1] More about object relational mapping: http://en.wikipedia.org/wiki/Object-relational_mapping

NOTE ON QCODO BACKWARD COMPATIBILITY

QCubed 2.1.0 **(released March 24, 2011)** is a complete rewrite of the client-side QCubed logic on top of jQuery.

Backward compatibility is maintained 100% on the server side, and mostly maintained on the client-side.

As of this writing on the site are present also the latest stable version 1.1.3. Please note that all releases in QCubed 1.0.x branch are fully backward-compatible with QCodo Beta 3, in order to assist the migrating QCodo community with a fully supported release for their production systems.

CHANGELOG AND BUG FIXES:

The following are the new features in this release (in the order of tickets in each group):

Javascript:

- Enable add External JS (#362)
- add low priority javascript (#635)
- Improve / Fix event trigger handling (use jQuery event functions) (#681)
- combine qcubed.js and control.js and get rid of _qc_packed.js (#805)
- Event/Action improvements: allow returning javascript objects/arrays/strings ... (#718)

Controls:

- New Control: Image Browser (#211)
- Create new QControl wrapper for jQuery Accordian (#462)
- user specified filter controls in the datagrid (#568)
- Periodic events triggering with the new QJsTimer control (#633)
- HtmlBefore and HtmlAfter are not honored in Render() (#661)
- QFloatTextBox,QIntegerTextBox : reduce code duplication and add a step interval property (#672)
- PbsQFileAsset Control (#711)
- Better integration of jQuery UI controls (#717)
- QSimpleTable - a new table control (#756)
- handle jQuery "property" events (#765)
- Make wrapper controls optional (#692)
- Ad an AddItems method to QListControl to add multiple items easily (#787)
- Execution of JavaScript before and after QXXXXAction on QControls (#788)
- QAccordion should track which item is open. (#790)
- Add HTML5 "search" input type (#791)
- Can't detect backspaces in fields (#723)

Databases and QQuery:

- Picking database columns for QQuery (#79)
- Query Profiler: add results of an EXPLAIN statement (#129)
- Proper Group By handling for databases other than MySQL (#270)
- optimization in InstantiateDbRow (#748)
- optimization in LogQuery (#751)
- optimization: minimize the array accesses in database adapters' GetColumn?() (#760)
- Adding DB-based session management (and form-state management) (#771)
- optimization: QueryCursor and InstantiateCursor methods (#759)

Codegen:

- Migrate Codegen Templates to PHP (#537)
- Feature request. Compability with php 5.3 (#646)
- optimization in InstantiateDbRow: avoid repeated string concatenations (#750)
- List Class names in AJAX drafts panel to show names in alphabetical order (#781)
- Modify codegen to create a list of icon names used by the JqUi buttons. (#804)

Other:

- add Last/First day of month function to QDateTime (#645)
- Make QApplication::GenerateQueryString more generic (#697)
- optimization: optional type argument for QDateTime (#757)
- EXPERIMENTAL - optimization: customizable behaviours for QType::Cast (#758)
- Integrating HTMLPurifier Library into QCubed (#766)
- Make number of items in form and panel draft lists configurable (#767)
- Integration of memcached support into QCubed (#770)

CHAPTER 1: INSTALLATION

1.1. PREREQUISITES

INTENDED AUDIENCE

This guide is intended for developers that are familiar that wish to understand how QCubed works to use QCubed to build their own applications.

We expect the reader to have some basic knowledge of system administration, and fair knowledge of database administration. Of course, as this is a PHP framework, so knowing PHP 5 and its object-oriented approach is a huge advantage.

WEB SERVER

The first thing you want to do is ensure that you have a standard, working installation of a webserver (e.g. Apache, IIS, etc.).

PHP 5

The framework is developed specifically for PHP5, and framework is not compatible with version 4 of PHP. The main reason is that PHP 5 has a completely redesigned, mature object model which QCubed takes direct advantage of.

DATABASE

You will need one of the following database platforms installed:

- MySQL (version 4 or 5)
- Postgres 8.0
- Microsoft SQL Server

You will need to have a username and a password to connect to the database and perform administrative tasks on that database.

1.2. INSTALLATION

WINDOWS

Installation instructions and scripts presented in this book are tested on Operating System Windows XP S.P.3 (installed on virtual vmware machine)

Note: the instructions in this tutorial are for a **WAMP5** installation. You may need to modify them slightly for a different configuration on your machine.

Details of installed environment are:

WAMP5 Version 2.2 http://sourceforge.net/projects/wampserver/files/WampServer 2/WampServer 2.2/

Tested browsers:

- Internet Explorer 6,7, 8
- Firefox 3.6.xx
- Seamonkey 2.9

Step 1: Downloading QCubed

QCubed can be retrieved from the framework download page: http://qcu.be/content/downloads.

Step 2: Unpacking QCubed

The file we have retrieved is a compressed zip file. Unzip it in temporary folder c:\temp.

This will create inside temporary folder the directory corresponding to QCubed version, with the following notable contents:

- ***assets***: this directory contains fonts,css, javascripts, examples, php scripts, plugins used to personalize your installation
- ***drafts***: this directory will contain generated drats panel as example to interact (list-create-edit) with db table
- ***includes***: this directory contains core module and all the other files that are required together with \assets to run QCubed.

Step 3: Moving files to their proper location

Now that we have retrieved and unpacked the files, we need to put the files on their proper location. The default location of the webroot on Apache in WAMP is ***c:\wamp\www***. **Please take attention that I installed wamp in drive d: so all reference to c:\ in example and description are changed to d:\ .** Verify this with your installation of Apache. The location is specified in the httpd.conf under the following setting:

```
DocumentRoot "D:/wamp/www"
```

To install QCubed, we need to copy the files from the wwwroot directory from the QCubed archive to this directory. As most of you will already have something in the default directory, we will do this in a subdirectory ***qcubed_210***, by issuing the following command from the location where we unpacked the QCubed-zip package

```
> move  c:\temp\2.1.0\  C:\wamp\www\qcubed_210\
```

Step 4: Setting permissions

This step is not necessary in a Windows environment.

Step 5: Setting the DOCROOT

The last step is to configure QCubed. The main configuration file for QCubed is located in the ***includes\configuration*** directory of QCubed and is named ***configuration.inc.php***

As we have installed QCubed in a subdirectory 'qcubed_202' on our webserver, we need to inform QCubed we have done this. Open this file using your favorite text editor, and locate the following section:

```
define ('__DOCROOT__', 'C:/xampp/xampp/htdocs');
define ('__VIRTUAL_DIRECTORY__', '');
define ('__SUBDIRECTORY__', '/qcubed2');
```

We will modify these 3 parameters to match our configuration:

- __DOCROOT__ : the system path in which the wwwroot folder of QCubed is installed.
- __VIRTUAL_DIRECTORY__ : only if your webserver is using virtual directories
- __SUBDIRECTORY__ : the subdirectory in which we installed QCubed as seen at the webserver level

In our configuration, set it to the following:

```
define ('__DOCROOT__', 'd:/wamp/www);
define ('__VIRTUAL_DIRECTORY__', '');
define ('__SUBDIRECTORY__', '/qcubed210');
```

STEP 6: Setting Database Parameters

We also need to set the database connection properties. This is done in the same ***configuration.inc.php*** file. Locate the line, and set the appropriate parameters for Server, Database, Username and Password, and encoding.

```
define('DB_CONNECTION_1', serialize(array(
          'adapter' => 'MySqli5',
          'server' => 'localhost',
          'port' => null,
          'database' => 'qcubed',
          'username' => 'root',
          'password' => '',
          'profiling' => false,
          'encoding' => 'utf8')));
```

If your configuration requires other database, set the adapter parameter to to one of the following:

- MySqli5 (MySQL v5.x, using the new mysqli extension)
- MySqli (MySQL v4.x, using the new mysqli extension)
- MySql (MySQL v4.x, using the old mysql extension)
- SqlServer (Microsoft SQL Server)
- PostgreSql (PostgreSQL) [2]

Also change line for timezone if required:

```
date_default_timezone_set('Europe/Rome');
```

If you want access the development components framework from other computers on the network – not just on the localhost - change this line in ***configuration.inc.php***. Note that setting this parameter to true in the production environment is not recommended.

```
define('ALLOW_REMOTE_ADMIN', true);
```

We should now be able to go to our webserver and view the QCubed start page by pointing in http://localhost/qcubed_210/. You can use localhost or machine name in that URL (in this book we will use magia so to test my installation I can point also to http://magia/qcubed_210/.

Once you do, you'll and see the QCubed start page at that URL:

[2] PostegreSql has some riserve word that conflict with example table name (please change user table name in users)

QCubed Development Framework 2.1 Development Release (QCubed 2.1)

Start Page

Welcome to QCubed!

If you are seeing this, the framework has been successfully installed. Say hi on QCubed Forum, we're here help you!

- Code Generator - to create ORM objects that map to tables in your database.
- View Form Drafts - to view the generated UI scaffolding (after you run the Code Generator).
- QCubed Examples - learn QCubed by studying and modifying the example files locally.
- Plugin Manager - to extend QCubed with community-contributed plugins.
- Update Checker - check for updates for QCubed core and plugins.
- QCubed Unit Tests - set of tests that QCubed developers use to verify the integrity of the framework. Test dataset required.
- QCubed Configuration Checker - monitors the health of your installation. Current status: all OK.

QCubed Settings

- QCUBED_VERSION = "2.1 Development Release (QCubed 2.1)"
- jQuery version = "1.7.1"
- jQuery UI version = "1.8.16"
- __SUBDIRECTORY__ = "/qcubed_210"
- __VIRTUAL_DIRECTORY__ = ""
- __INCLUDES__ = "D:/wamp/www/qcubed_210/includes"
- __QCUBED_CORE__ = "D:/wamp/www/qcubed_210/includes/qcubed/_core"
- ERROR_PAGE_PATH = "/qcubed_210/assets/_core/php/error_page.php"
- PHP Include Path = ".;C:\php\pear"
- QApplication::$DocumentRoot = "D:/wamp/www"
- QApplication::$EncodingType = "UTF-8"
- QApplication::$PathInfo = ""
- QApplication::$QueryString = ""
- QApplication::$RequestUri = "/qcubed_210/assets/_core/php/_devtools/start_page.php"
- QApplication::$ScriptFilename = "D:/wamp/www/qcubed_210/assets/_core/php/_devtools/start_page.php"
- QApplication::$ScriptName = "/qcubed_210/assets/_core/php/_devtools/start_page.php"
- QApplication::$ServerAddress = "192.168.0.21"
- QApplication::$Database[1] settings:
  - adapter = 'MySqli5'
  - server = 'localhost'
  - port = NULL
  - database = 'qcubed'
  - username = 'root'
  - password = 'hidden for security purposes'
  - caching = false
  - profiling = false
  - encoding = 'utf8'

QCUBED

For more information, please visit the QCubed website at http://www.qcu.be/
Questions, comments, or issues can be discussed at the Examples Site Forum

Or this page to assist you if configuration file info needs further modification:

Welcome to QCubed!

PHP5 Model-View-Controller framework

This simple wizard will help you configure QCubed for first use. It'll take you just a couple minutes. If you have any questions along the way, feel free to ping us on the support forums, a vibrant community is there to help you all the time. There's also a chat room where you can get help right away.

Here's what you need to do:

1. Set the __SUBDIRECTORY__ constant in /includes/configuration/configuration.inc.php. Most likely value: "/qcubed_210"

[I'm done, continue]

Ignore these warnings and continue (not recommended)

LINUX

To install on Linux you can use same instructions as above, adapting directory names to Unix installation and setting permissions. This means that steps 4 and 5 have to be different from above; everything else is the same.

Note: the distribution used for this tutorial is Fedora Core 4 with Apache httpd-2.0.54-10.3 and PHP php-5.0.4-10.5.

STEP 4: SETTING PERMISSIONS

Because the code generator generates files in multiple locations, you want to be sure that the webserver process has permissions to write to the DOCROOT.

The simplest way to do this is just to allow full access to the docroot for everyone. While this is obviously not recommended for production environments, if you are reading this, I think it is safe to assume you are working in a development environment.

On Unix/Linux, simply run

```
#  chmod  -R  ugo+w  /var/www/html/Qcubed210
```

STEP 5: SETTING THE DOCROOT_SUBFOLDER AND DATABASE PARAMETERS

The last step is to configure QCubed by editing ***includes\configuration\configuration.inc.php.***

As we have installed QCubed in a subdirectory 'QCubed' on our webserver, we need to inform QCubed we have done this. Open this file using your favorite text editor, and locate the following section:

```
define ('__DOCROOT__', '/home/QCubed/wwwroot');
define ('__VIRTUAL_DIRECTORY__', '');
define  ('__SUBDIRECTORY__',  '');
```

In our configuration, set it to the following (you might need to modify the paths below to match your web server installation):

```
define ('__DOCROOT__', '/var/www/html/QCubed/wwwroot');
define ('__VIRTUAL_DIRECTORY__', '');
define  ('__SUBDIRECTORY__',  '/Qcubed210');
```

Chapter 2: Introduction to QForms

2.1. Definition

QForms is a user interface library built into QCubed. As much as everybody likes the code generation of QCubed, an equally important piece of QCubed is QForms. So what are QForms?

> ***"QForms is an object-oriented, stateful, event-driven architecture for forms rendering and handling"***

Nice... but what does it mean?

2.2. Our first QForms application

Let's explore QForms by creating a sample application: a page with a button. The page will display how many times the button has been clicked.

A QForm based page consists of 2 files. For our first application, these will be

- ***example1.php***, contains the **logic** of our application
- ***example1.tpl.php***, contains the **presentation**layer of our application

So, to create a basic QForm application, we need to create those 2 files. Let's do so:

example1.php:

```
<?php

// Load the Qcubed framework
require('includes/configuration/prepend.inc.php');

class Example1 extends QForm {
   protected function Form_Create() {

}
}

Example1::Run('Example1');

?>
```

example1.tpl.php:

```
<html><head>

<title>Our first QCubed sample</title></head>

<body>

<?php $this->RenderBegin(); ?>
```

```
<?php $this->RenderEnd(); ?>

</body></html>
```

EXAMPLE BREAKDOWN

Let's take a closer look at example1.php line by line:

```
require('includes/configuration/prepend.inc.php');
```

This is the line which tells the PHP page to load the QCubed framework. We will need to include this in every page where we want to use QCubed.

```
class  Example1  extends  QForm  {
...
}
```

We are going to create a QForm based application, so we create a new class called “Example1”. We derive it from QForm, the parent of all forms.

```
protected  function  Form_Create()  {
  ...
}
```

The function ***Form_Create()*** is called the first time our page is loaded. We will have to put our initialization code here. For now, we have nothing there.

```
Example1::Run('Example1');
```

This calls the parent class (remember, it’s QForm) ***Run()*** method, causing the framework to run the page.

This ends our review of the example1.php page.

The other file, example1.tpl.php, you can see that the file contains mostly HTML. The only dynamic, non-HTML pieces the following:

```
<?php  $this->RenderBegin();  ?>

<?php  $this->RenderEnd();  ?>
```

This is necessary to allow the Qform framework to manage state.

Now, if we would fire up our browser, and load the ***example1.php*** page, this does not show us much, does it? Let's add some things (only **title** is present to confirm that our script is living).

ADDING CONTROLS AND STATE

First, think of Example1 as your application. Your application holds several other objects and variables. For this application, we will add 3 things: a variable to hold state and 2 controls (a label and a button)

To do this, add the following to ***example1.php***:

```
...

class  Example1  extends  QForm  {
  protected $intNumberOfClicks;
protected $lblNumberOfClicks;
protected $btnButton;

protected  function  Form_Create()  {

...
```

This defines the items we will use in our application: an integer, a label and a button.

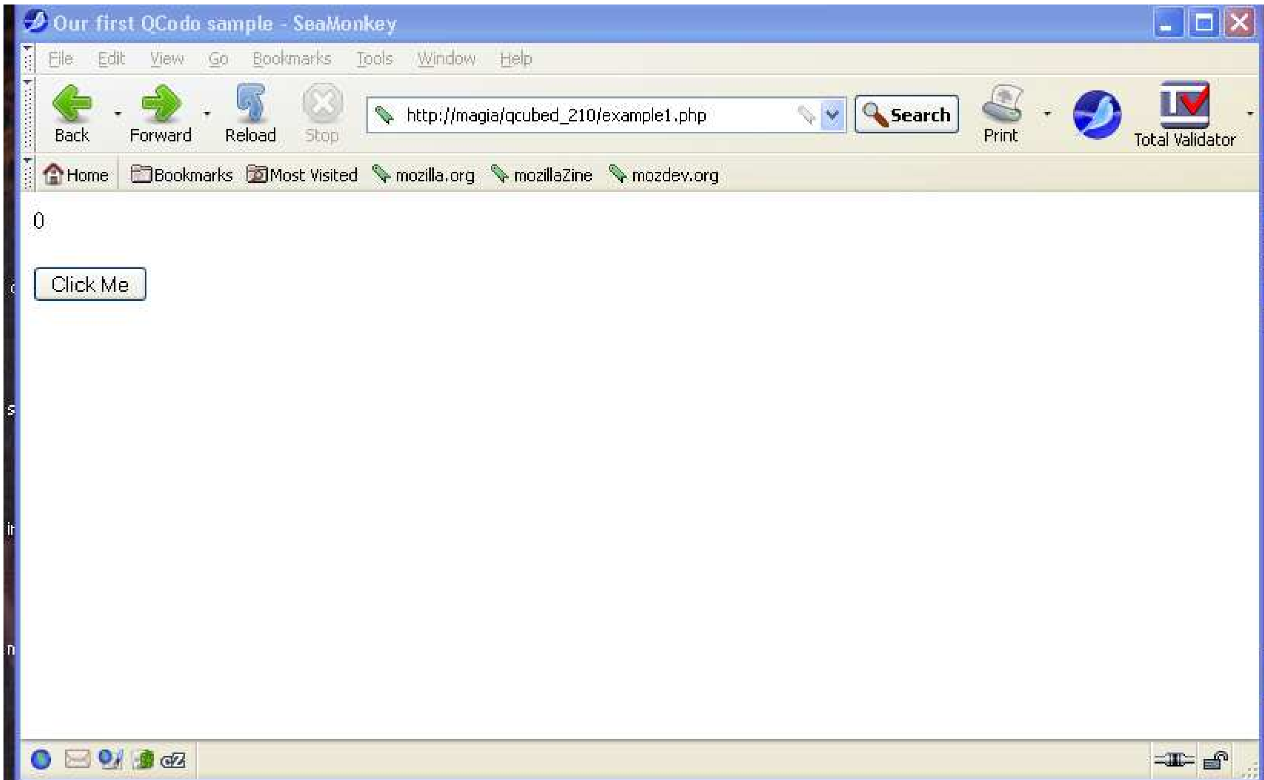
Next, we will assign some default values to it. As explained above, we need to do this in the ***Form_Create()***, as this is where the initialization of our form happens:

```
protected function Form_Create() {
$this->intNumberOfClicks = 0;
$this->lblNumberOfClicks = new QLabel($this);
$this->lblNumberOfClicks->Text = $this->intNumberOfClicks;
$this->btnButton = new QButton($this);
$this->btnButton->Text = "Click Me";
}
```

We also want to display the button and the label. We do this by calling the ***Render()*** function of the objects we want to display in ***example1.tpl.php***:

```
...

<?php  $this->RenderBegin();  ?>
<?php  $this->lblNumberOfClicks->Render();  ?>
<br/><br/>
<?php  $this->btnButton->Render();  ?>
<?php  $this->RenderEnd();  ?>

...
```

Fire up that browser, and load the page:



This already *shows* something, but does not really *do*anything. So let's add some events and event handling logic to take care of the "doing stuff" part.

ADDING EVENTS

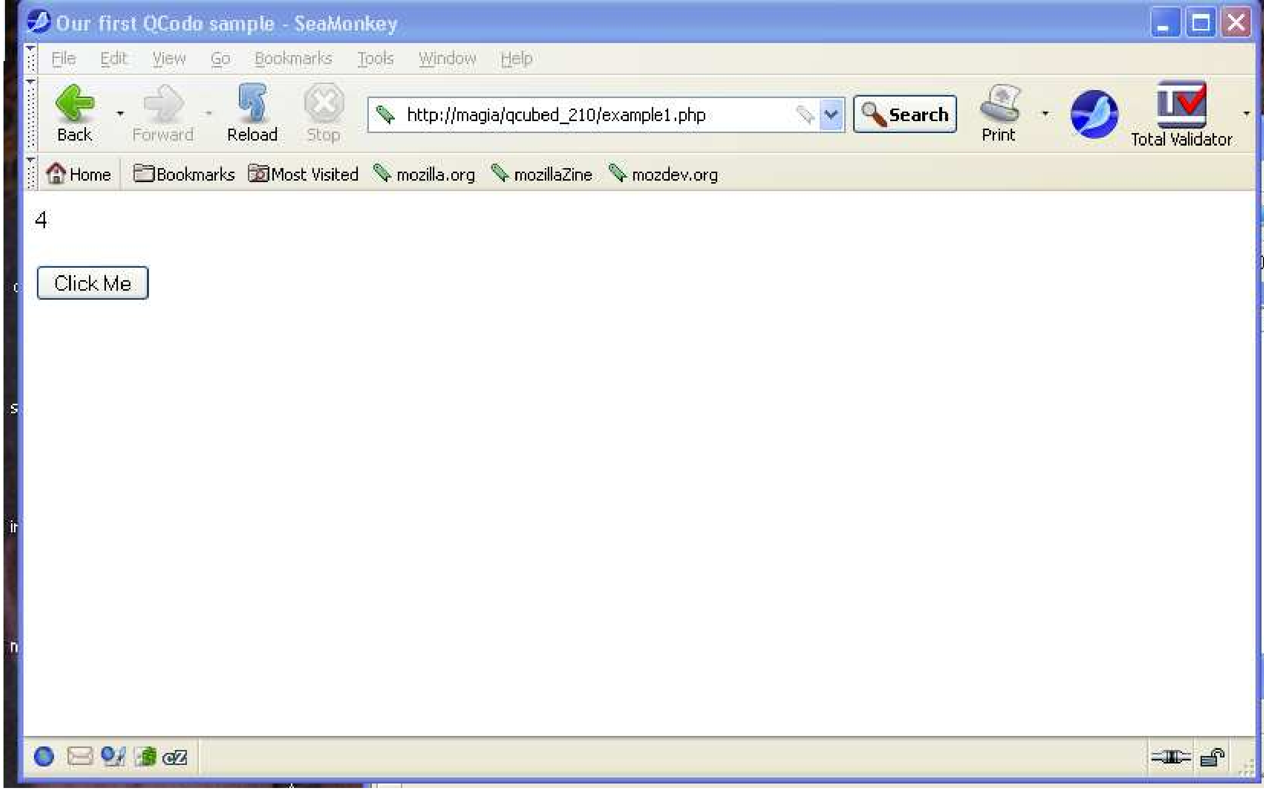
Now we have the "event-driven" part: when we click the button, we want the form variable ***$intNumberOfClicks*** to be incremented, and the display to be updated. What we need to do is to assign an event to $btnButton. So in the Form_Create(), we add:

```
...
$this->btnButton->Text  =  "Click  Me";
$this->btnButton->AddAction(new QClickEvent(), new QServerAction('btnButton_Click'));

...
```

This tells the application that we want to have a new QServerAction. The function that will be executed is ***btnButton_Click()***. So we also need to create this function:

```
protected function btnButton_Click($strFormId,  $strControlId,  $strParameter)  {
$this->intNumberOfClicks++;
$this->lblNumberOfClicks->Text  =  $this->intNumberOfClicks;
}
```

Save and refresh the page. When you click the button, you should now see the counter increment.



SUMMARY

So, let us review the definition we started with:

> ***"Qforms is an object-oriented, stateful, event-driven architecture for forms rendering and handling"***

- object-oriented: we created our own object "example1", assigned other objects to it, and assign them values and events. Note that the object model also applies to the data objects, which we will cover later.
- stateful: notice that QCubed "remembers" the value of the intNumberOfClicks. We did not have to store it in a session or database ourselves
- event-driven: make objects listen to events (such as in our example a click event), and execute some code in response to the registered events.
- forms rendering and handling: notice that we did not have to code any <form> or <input> tags. QCubed handles this for us.

To summarize, to create a QForms application you need to

- create 2 files: one for your logic and one for your presentation
- in your .php file, create a new class derived from QForms. Add the objects you need for your application to it, assign default values and actions to it, and create the functions to handle your actions
- in your .tpl.php file, define the layout, call ***RenderBegin()*** and ***RenderEnd()***, and for each control you need to display, call the ***Render()*** function.

CHAPTER 3: INTRODUCTION TO CODE GENERATION

WHAT QCUBED DOES FOR YOU

In short, the QCubed Code Generator takes in your database schema, and outputs:

- Data Objects (Model classes) mapped to your database tables
- HTML Form Drafts, or the scaffolding for the user interface that handles the Object to HTML mapping via the use of QForms

CODE GENERATION OF DATA OBJECTS

For each table in your database, QCubed will generate the code to translate the table to an object that contains all the basic CRUD-type methods to Create, Restore, Update, and Delete data to/from the database. In addition to the basic CRUD functionality, QCubed also generates complex methods to retrieve by index, associate and unassociated related objects, and perform early- and late-binding on those related objects.

HTML FORM DRAFTS

For each table in your database, QCubed will also generate code-generated implementations of the QForm class, for the PHP front-end HTML code to view, create, edit and delete your table objects.

WHAT YOU STILL HAVE TO DO

After the code generation has been performed, you will be have basic CRUD functionality for all your tables. You will also have objects that represent the table structure which will allow you to easily access the data in your database.

Now it's up to you to use the created objects and/or to modify and extend the generated form drafts to match your business logic.

EXAMPLE

In the second example, we will create a database-drive application to manage a todo list using only the code-generator tool.

DATABASE LAYOUT

Note: the best way to proceed from here is to create a separate database schema to put in the tables from this chapter. Also modify the configuration.inc.php file in the includes directory in order to connect to the proper schema (db name in this example is test).

```
define('DB_CONNECTION_1', serialize(array(
'adapter' => 'MySqli5',
'server' => 'localhost',
'port' => null,
'database' => 'test',
'username' => 'root',
'password' => '','profiling' => false,
'encoding' => 'utf8' )));
```

Our simple todo list application has a single table with 2 fields: the id and a description.

Note: to create this table you can use phpMyAdmin that comes installed in Wamp.

```
CREATE TABLE `todolist` (
`id` int(11) NOT NULL auto_increment,
`description` varchar(100) NOT NULL,
PRIMARY KEY (`id`)
) ENGINE=InnoDB
```

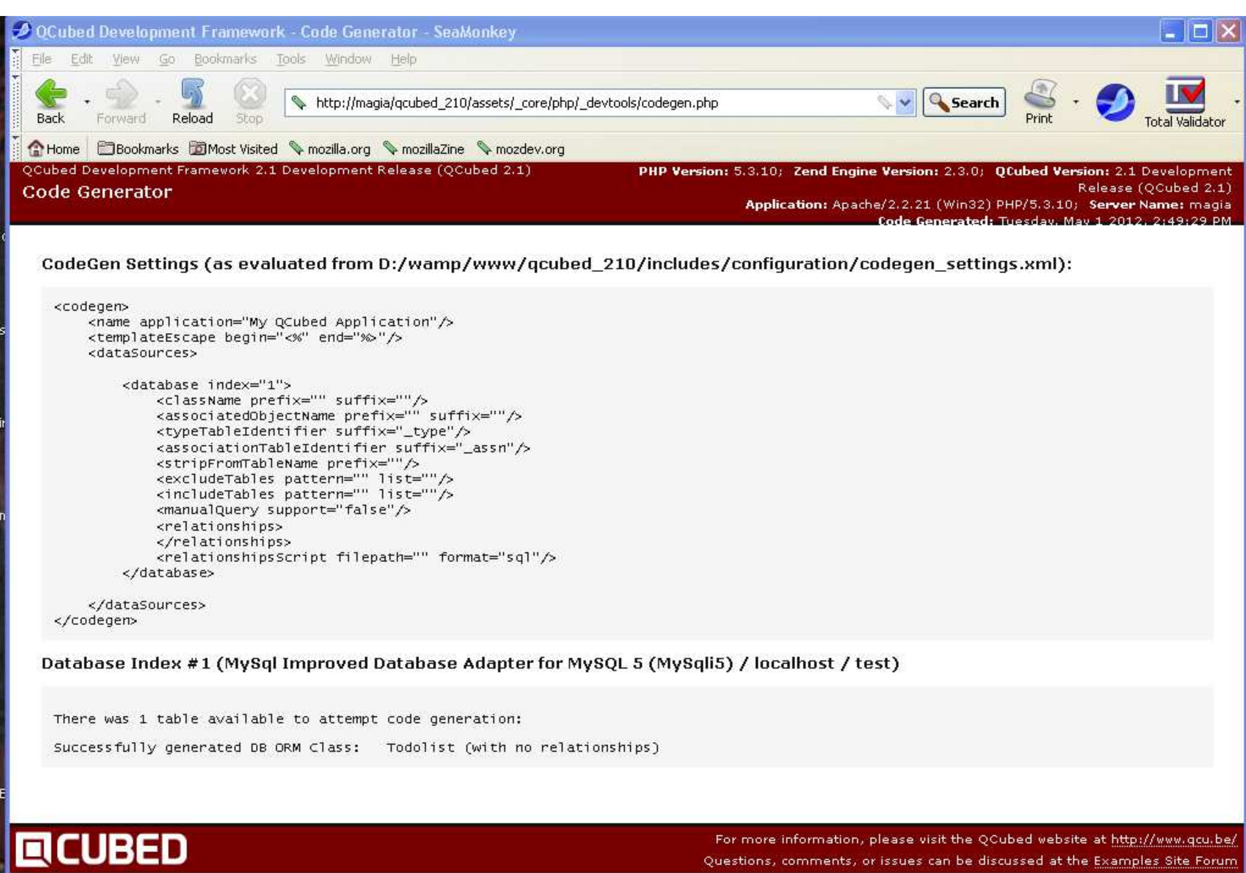
CODE GENERATION

To use the code generation capabilities from QCubed, simply open your browser, and go to the homepage of your QCubed installation.

You should find 3 links there:

- Code Generator - to create ORM objects that map to tables in your database
- View Form Drafts - to view the generated UI scaffolding (after you run the Code Generator).
- QCubed Examples - learn QCubed by studying and modifying the example files locally.

To generate the code for, click the first link.



If all goes well, you should see a successfully result page. The page also provides an overview of the tables for which code generation has been performed.

OVERVIEW OF THE GENERATED FILES

For each table in the database (in this case there is only one called "todolist") the code generator generates 16 files:

1. includes\meta_controls\TodolistDataGrid.class.php
2. includes\meta_controls\TodolistMetaControl.class.php
3. includes\meta_controls\generated\TodolistDataGridGen.class.php
4. includes\meta_controls\generated\TodolistMetaControlGen.class.php
5. includes\model\Todolist.class.php
6. includes\model\generated\TodolistGen.class.php
7. includes\formbase_classes_generated\TodolistEditFormBase.class.php
8. includes\formbase_classes_generated\TodolistListFormBase.class.php
9. drafts\todolist_edit.php
10. drafts\todolist_list.php
11. drafts\todolist_edit.tpl.php
12. drafts\todolist_list.tpl.php
13. drafts\panels\TodolistEditPanel.class.php
14. drafts\panels\TodolistListPanel.class.php
15. drafts\panels\TodolistEditPanel.tpl.php
16. drafts\panels\TodolistListPanel.tpl.php

Additionally, .these global files are created or updated:

17. includes\model\generated_class_paths.inc.php
18. includes\model\generated_type_class_paths.inc.php
19. includes\model\generated\QQN.class.php

This seems like a lot. But let's break it down into what we already know QCubed is doing for us:

- Code Generation of Data Objects
- the HTML Form Drafts

The files 1-6 in the list are the result of the Code Generation of the Data Object, DataGrid and Controls for our Todo list and the remaining lines are for the HTML form drafts to allow us to do the basic CRUD functionalities on the objects.

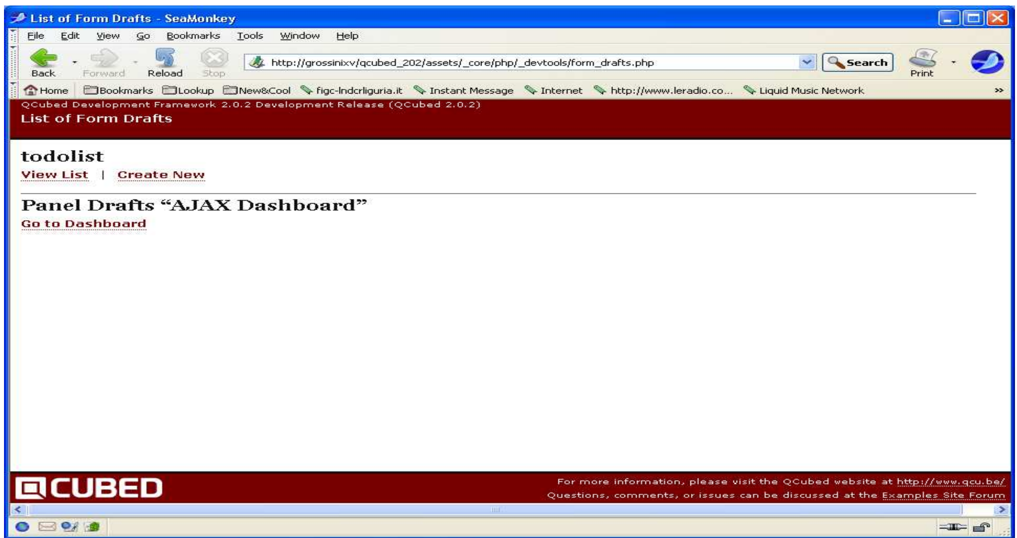
VIEWING THE RESULT

To see the result of the code generation, go back to the homepage of your QCubed installation and now, click the second link, ***drafts***.

Please note that in directory \drafts and \drafts\dashboard from installation are already present:

- \drafts\index.php
- \drafts\panels\index.php

It will show the following page:



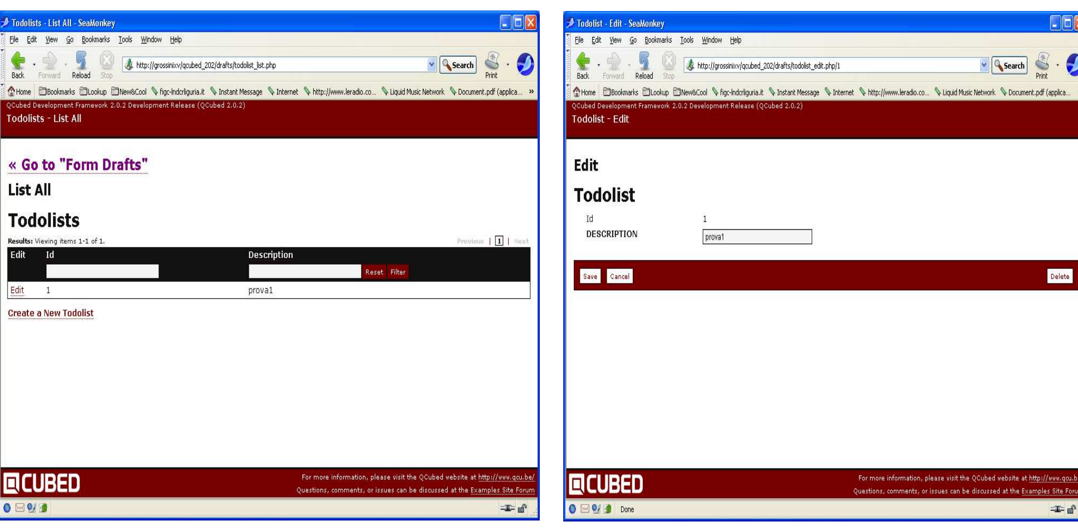


Illustration 1: Todolist List

Illustration 2: Todolist Edit

This page will show us a list of all the tables for which code was generated. For each table, you can view the list, or edit or create a new entry for the table.

Go ahead, and play around with the table. You can add todo items, delete items(through the edit page) and modify items. You can also sort by description and id, and if your list grows, pagination is also included

Chapter 4: Understanding the generated code

Now that we have learned how to generate code from an existing database model using Qcubed, we can go deeper into the details of the files and code that have been generated.

Data Objects

For data objects, only two files are generated. ***Todolist.class.php*** (in includes\models) and ***TodolistGen.class.php*** (in includes\models\generated).

TodolistGen.class.php contains all the basic CRUD-type method to create, restore, update, and delete your data to/from the database. These functions are:

- Load($intId, $objOptionalClauses = null)
- LoadAll($objOptionalClauses = null)
- CountAll()
- Save($blnForceInsert = false, $blnForceUpdate = false)
- Delete()
- DeleteAll()
- Truncate()
- Reload()

The TodolistGen.class.php also contains code to allow access the fields in the table.

For each field, the code generated created properties with the same name as the field in the table. In our case, the code generator created the following properties:

- Id
- Description

Later on, you will see that ***TodolistGen.class.php*** also provides more complex methods to retrieve by index, associate and unassociated related objects, and perform early- and late-binding on those related objects.

But for now, just realize that ***TodolistGen.class.php*** is the generated object relational model for the todolist table.

The ***Todolist.class.php*** file is more or less a blank Todolist class that extends the ***TodolistGen*** class. Throughout the system, calls to manipulate To Do List entries should be done on the ***Todolist*** class and not the ***TodolistGen*** class (and in fact, doing so will throw an error because TodolistGen is an abstract class).

This design is to allow you as a developer to write custom functionality, implement business logic, etc. in ***Todolist.class.php***, but still allow you to re-code generate as often as possible, without fear of losing any of your customizations, changes, etc.

> ***Remember: everything in a "generated" folder will always be recreated on subsequent calls to the code generator***

QForm and QPanel - Drafts and Dashboard

The remaining 12 files are code-generated implementations of the QForm class, for the PHP front-end HTML code to view, create, edit and delete Todolist objects from the system.

2 types of frontends are created:

- a basic form_drafts frontend
- an advanced ajax panels drafts frontend.

To display the form_drafts pages, 4 files are used: Two of the files are used for the "List All Todolist" page, and the other two files are used for the "Edit a Todolist" page.

To display the dashboard panel_drafts pages, another 4 files are created.

If you take a look at the draft pages, you will notice that these are in fact QForms: the list

and edit .php files (controllers) each have a corresponding .tpl.php file (views).

- `drafts\todolist_edit.php`
- `drafts\todolist_list.php`
- `drafts\todolist_edit.tpl.php`
- `drafts\todolist_list.tpl.php`
- `drafts\panels\TodolistEditPanel.class.php`
- `drafts\panels\TodolistListPanel.class.php`
- `drafts\panels\TodolistEditPanel.tpl.php`
- `drafts\panels\TodolistListPanel.tpl.php`

The dashboard is an advanced code generated, ajax powered way of providing CRUDfunctionality. Instead of using separated QForms, this application uses QPanels, whichhave been generated for you during the code generation.

Meta Controls

Meta controls are something relatively new to QCubed. In essence, they allow you to rapidly set up the controls that you will use in your forms. When making controls for database fields, you will find that many times the process is the same: create a control, give it a name, and add the corresponding data from the field. The QCubed generation speeds this process by creating functions which will do perform these tedious actions for you. Here is an example. Don't worry about understanding WHAT all the code does for now. The more important thing to notice is typing involved. In order to create a text box for the Description field of our Todolist table, we would type something like:

```
$this->txtDescription = new QTextBox($this);

$this->txtDescription->Name = QApplication::Translate('Description');

$this->txtDescription->Text = $this->objTodolist->Description;

$this->txtDescription->Required = true;

$this->txtDescription->MaxLength = Todolist::DescriptionMaxLength;
```

With meta controls, this is placed into the meta control class as a method. So, instead ofthe above lines, you would type (mctTodolist is the Todolist Meta control object):

```
$this->txtDescription = $this->mctTodolist->txtDescription_Create();
```

And that is it. Clearly, you can see the advantage if your page contains numerous controls. The best part of these controls and QCubed itself is that you don't have to use them. You have the freedom to not use the meta controls if they do not fit your needs. Need a different label for the Name? No problem, just set up your control manually. Or maybe you could write a new method for the class in `the meta_controls` folder. It is your choice what you use and what you do not. You are not tied to certain features with this code generation.

META CONTROLS DATAGRID

And for ***todolist_list.php*** similar magic functionality is added by the ***TodolistDataGrid*** and the generated ***TodolistDataGridGen.class.php***. In ***todolist_list.php*** the only activity required to you is choose what field you want include in list, if you want pagination and how many items per page, styles etc. The low level coding work is done for us by code generator.

- `includes\meta_controls\TodolistDataGrid.class.php`
- `includes\meta_controls\TodolistMetaControl.class.php`
- `includes\meta_controls\generated\TodolistDataGridGen.class.php`
- `includes\meta_controls\generated\TodolistMetaControlGen.class.php`

Now, in reality, because QForm pages are really only just two files (see chapter 2), QCubed could have simply generated a `todolist_edit.php` file and a `todolist_edit_tpl.php` file to have a draft "Edit" page, where the .php file has the display logic and the .tpl.php file be the HTML template.

But the Code Generator basically splits `todolist_edit.`php into two files:

- \drafts\todolist_edit.php is a TodolistEditForm class
- \includes\formbase_classes_generated\TodolistEditFormBase.class.php, which is a TodolistEditFormBase class

This is definitely a key value of the QCubed code generator and the way that it interacts with creating QForms. It allows you a great starting point to help you design out the pages/screens for your application.

SUMMARY

Files in “generated” directories will always be overwritten. Any customizations should therefore not be done in the generic of base classed, but in the subclasses of these objects.

QCubed generates Data objects, form and panel (in panels) drafts for basic CRUD functionality.

CHAPTER 5: MORE ON CODE GENERATION

In the previous chapter, you have learned how to use the code generator of QCubed, and what files are generated by QCubed. In this chapter, you will take this one step further by using QCubed to create an simple time tracking application. We will do so by creating and gradually extending our database model to show how QCubed behaves on the changes.

CREATING THE DATABASE

Before we can start, we need to create a database. In the beginning of this chapter, our database will contain only 2 tables. Later, we will extend the database model with more tables and extending the existing tables with additional fields as we need them.

Note: it is again a good idea to create a new schema for this chapter (timetrack) and remove the scripts createt in chapter 4 by reinstalling Qcubed from scratch saving only work done in configuration .inc.php**. Also, modify the database connection parameters in configuration.inc.php in order to connect to the correct schema**.

In a first phase, we will want to have users and projects for which we want to time tracking. So let us create the tables:

```
CREATE  TABLE  `user`  (
  `id` int(11) NOT NULL auto_increment,
  `firstname` varchar(100) NOT NULL default '',
  `lastname` varchar(100) NOT NULL default '',
  `email` varchar(100) NOT NULL default '',
  PRIMARY KEY (`id`)
  )  ENGINE=InnoDB  ;

CREATE  TABLE  `project`  (
  `id` int(11) NOT NULL auto_increment,
  `name`  varchar(100)  NOT  NULL  default  '',
  `manager`  int(11)  NOT  NULL  default  '0',
  PRIMARY KEY (`id`),
  KEY  `fk_user_manager`  (`manager`)
  ) ENGINE=InnoDB ;
ALTER TABLE `project`
  ADD CONSTRAINT `project_ibfk_1` FOREIGN KEY (`manager`) REFERENCES `user` (`id`);
```

As you can see, we have created 2 tables. The tables are called “user” and “project”.and we added relation.

FOREIGN KEYS

When performing code generation, in addition to provide basic CRUD functionality, QCubed will also analyze the Foreign Key relationships in your database and generate relationships between your objects.

Whenever your table has a column which is a Foreign Key to another table, the dependent class (the table with the FK) will have an instance of the independent class (the table where the FK links to).

So in our database, we have a manager column in our project table, which is a foreign key to the user table. It will be used to hold the manager of our project.

This results in a ManagerUser property (of type User) in our Project class.

MyISAM and foreign keys

By default, the MySQL table type is MyISAM. Unfourtunately, MyISAM does not have support for foreign keys. In order for QCubed to use the Foreign Keys relationships, the table type must therefore be InnoDB.

Luckily, QCubed also provides a way around this. Please take a look at the following Forum message from Mike in order to overcome this problem

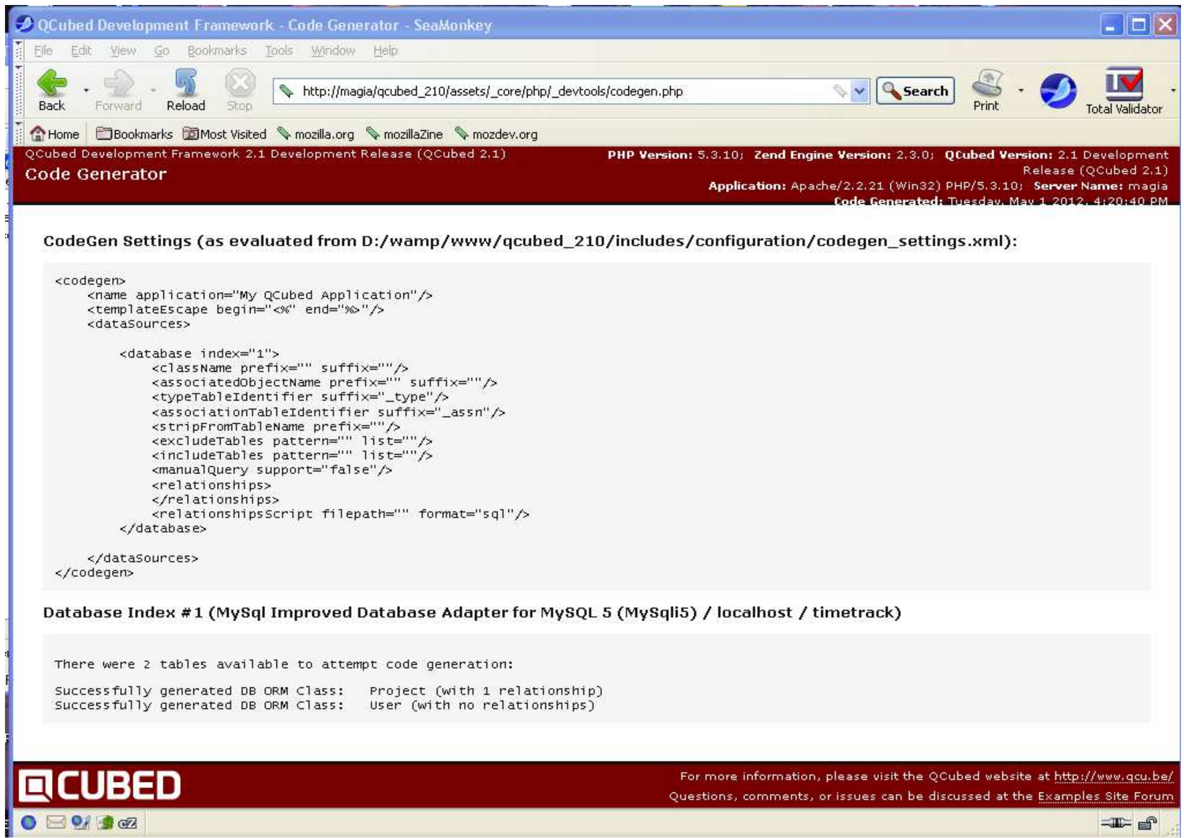
http://www.qcodo.com/forums/topic.php/3/1/0/?strSearch=innodb

On the examples site you can find an example that should answer all your questions. It is located in Section 1 (Basic CodeGen) under the name 'defining relationships without foreign keys'.

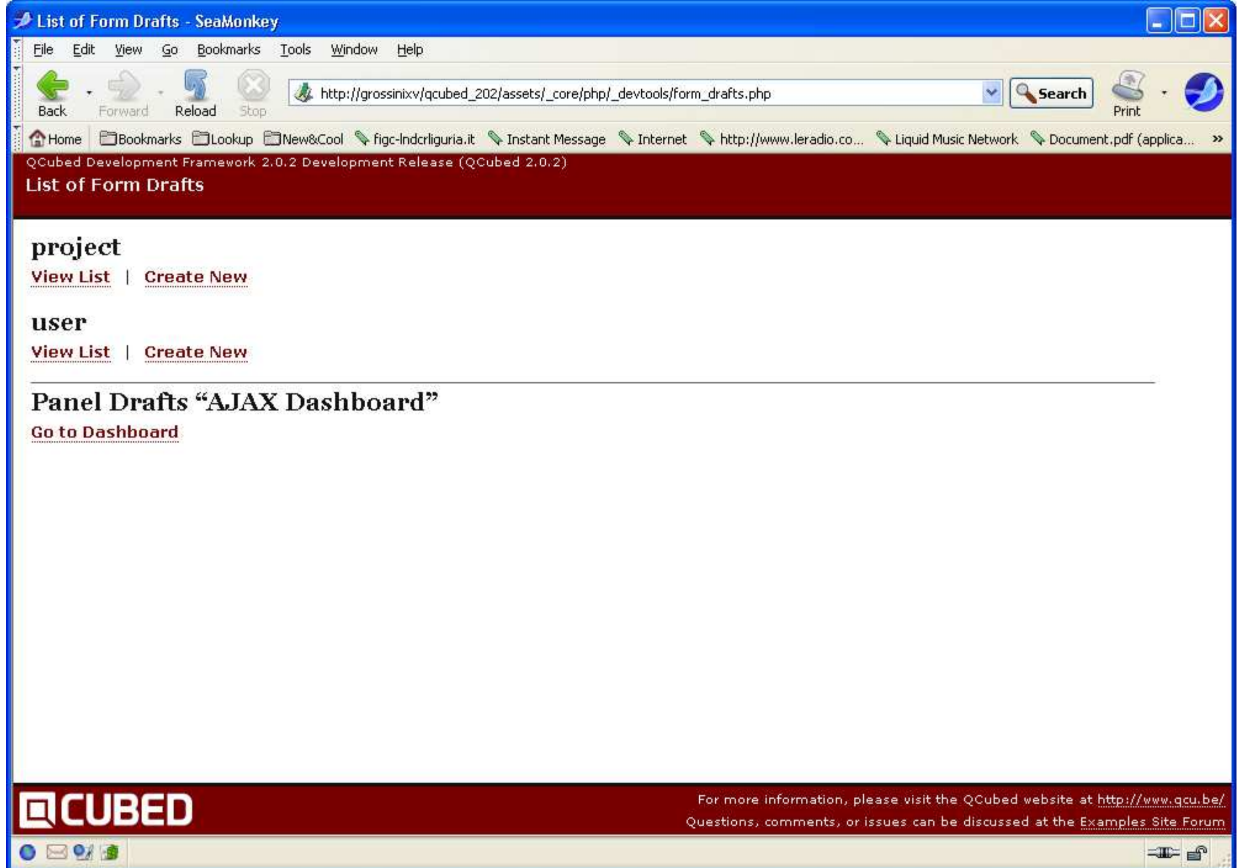
Code generation – step 1

Let's see what QCubed does with these tables. First, as this is a new project, make sure that you connect to the correct database with QCubed. As you know, you can change the database connection parameters in configuration.inc.php which is in the includes directory.

Go to your main page of QCubed and click the codegen link. You should see something like the following screen:



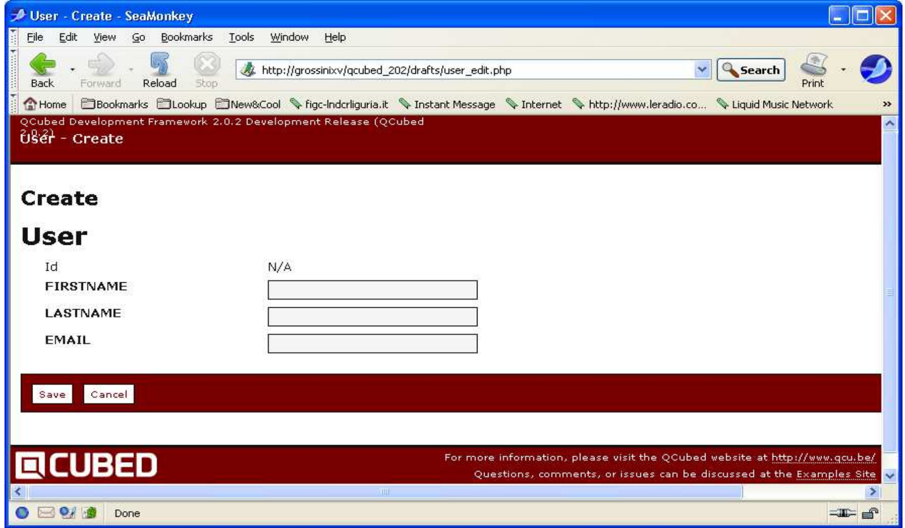
QCubed has generated code for the 2 tables we have. So... let's go to them, and see what they look like: go back to the main page, and select the form drafts link.



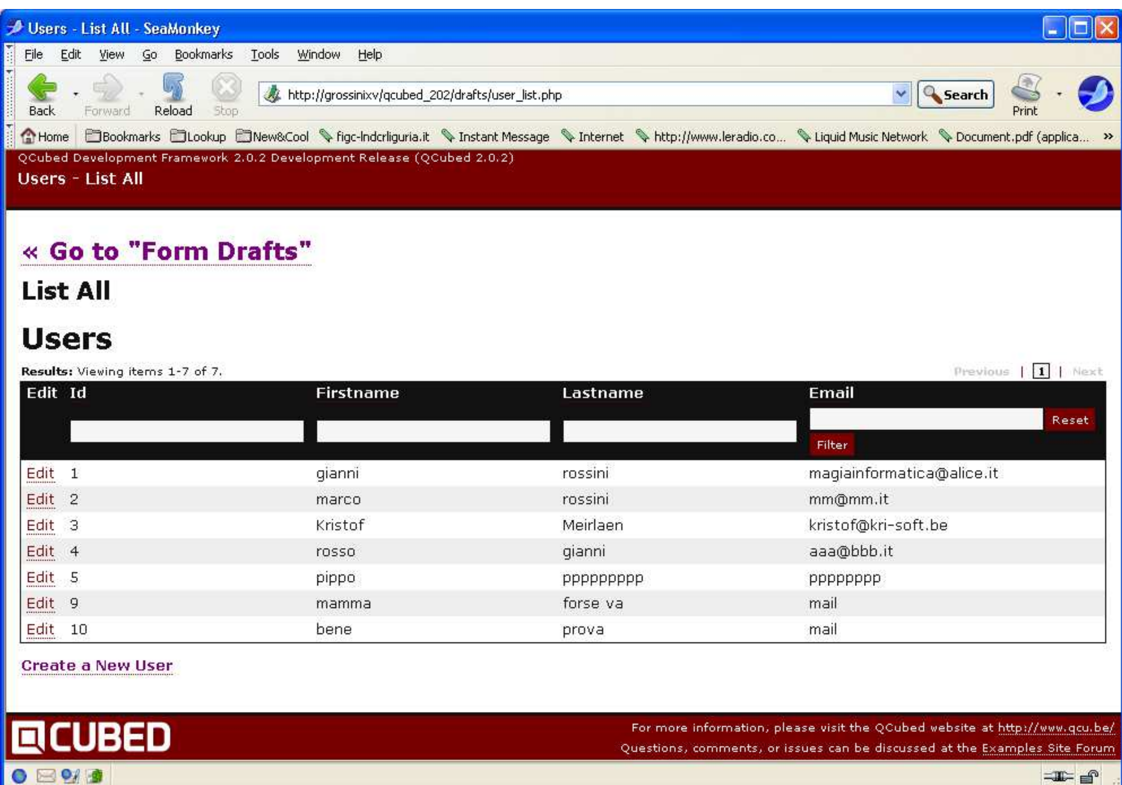
This should also be familiar, as we have seen in the previous chapter that QCubed will have generated forms for each of our table.

ADDING DATA

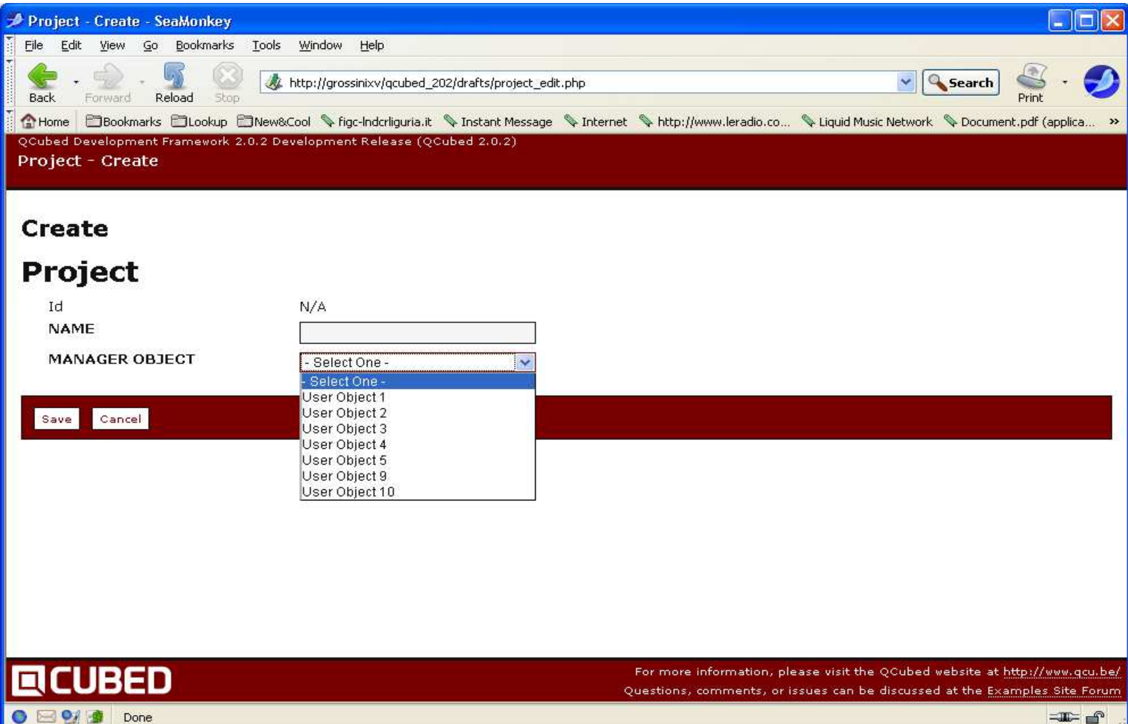
Go on, and create a few users now. Click on the create a New User link and insert some random data.



We can see or edit user created using view list:



After we have created some users, we also would like to create some projects:



Notice that instead of displaying a name, the framework returns the “Object”. This is because QCubed does not know how you want the object to be displayed.

OVERRIDING THE DEFAULT RETURN STRING FOR AN OBJECT

Remember that for each table 16 files are created: 2 files for the data objects and 14 files for the form and panel drafts.

To change the default return string for a data objects, we will have to change one of the data object files. As one of them resides in the `model/generated` folder, we don't want to use this one, as when we would re-code generate, the changes would be overwritten.

Therefore, to change the default return string for an object, locate the object.class.php file in the `includes\model` directory. In our case, we want to change the user object, so open up User.class.php.

User.class.php contains the class User and this class is inherited from UserGen.

Inside User.class.php, there is currently one function present:

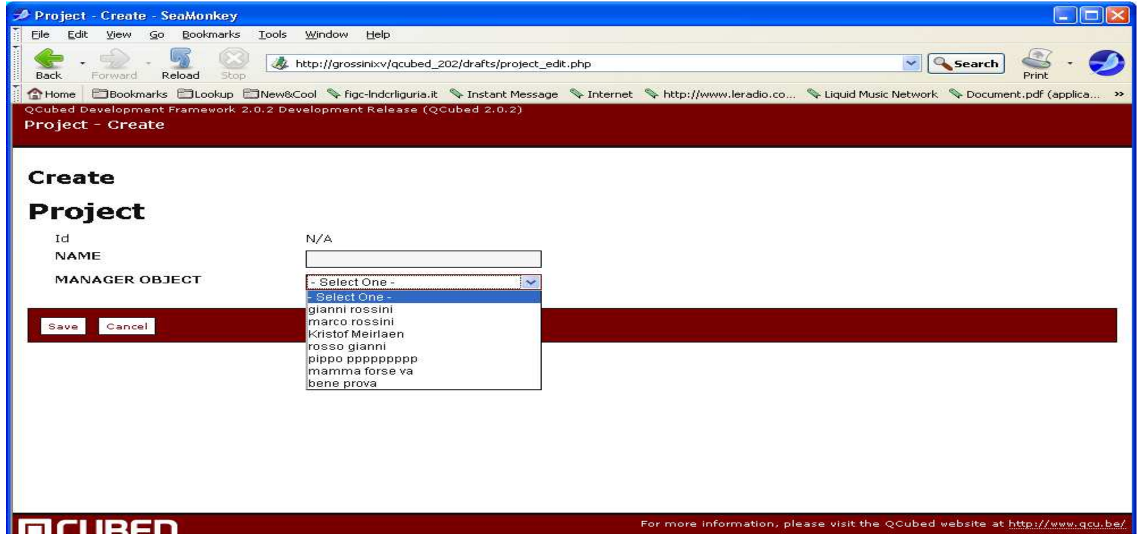
```
public function __toString() {
return sprintf('User Object %s',   $this->intId);
}
```

As you can see, by default QCubed returns “User Object “ and the ID of the row, which is what we saw when trying to create a project.
To change it, simply overwrite the return statement with something like:

```
public function __toString() {
return sprintf('%s %s',   $this->strFirstname, $this->strLastname);
}
```

Here, “$this” points to the current object (User object extended from UserGen object). The strFirstname and strLastname are the (protected) properties of this object that contain the data from the Firstname and Lastname fields.

Save User.class.php, and reload the page to add a project. As you can see, the names are now displayed correctly:



Go ahead, and add some projects. We will need them in our next step.

EXTENDING THE DATABASE

Now that we have created users and projects, we should be able to start some time tracking. For this, we obviously need a table to hold the tracking data:

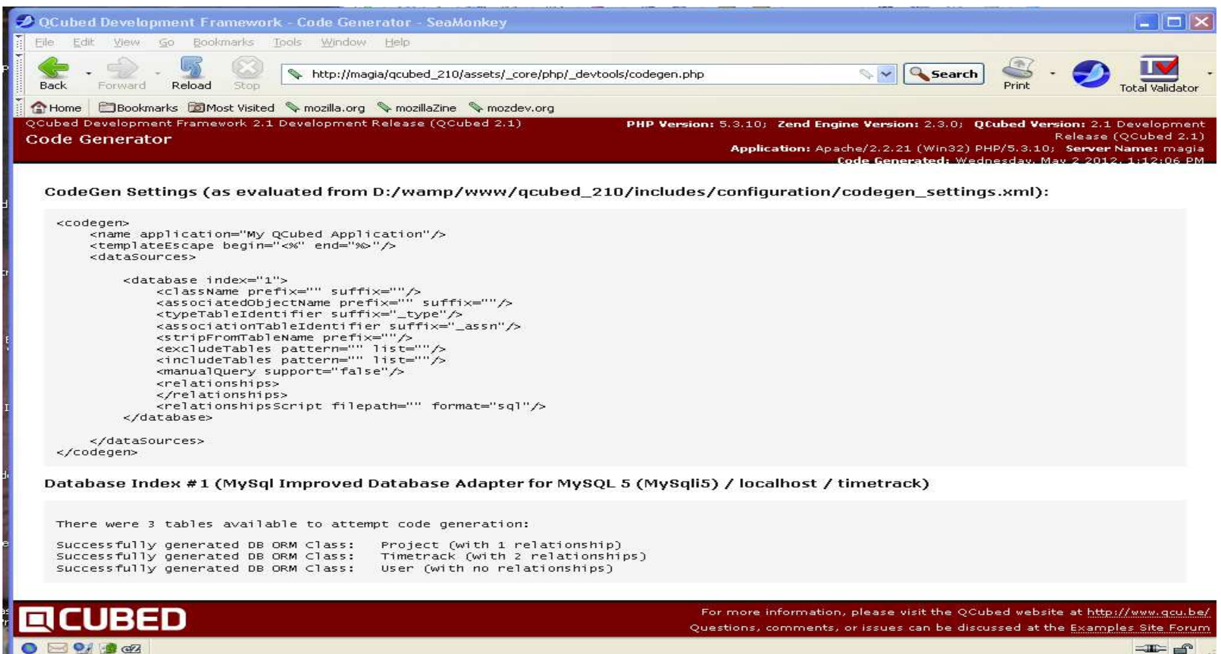
```
CREATE  TABLE  `timetrack`  (
`id` int(11) NOT NULL auto_increment,
`user` int(11) NOT NULL default '0',
`project` int(11) NOT NULL default '0',
`date` date NOT NULL default '0000-00-00',

`start_time` time NOT NULL default '00:00:00',
`end_time` time NOT NULL default '00:00:00',
PRIMARY KEY (`id`),
KEY `fk_project_project` (`project`), KEY `fk_user_user` (`user`),

CONSTRAINT  `fk_user_user`  FOREIGN  KEY  (`user`)  REFERENCES  `user`  (`id`),
CONSTRAINT `fk_project_project` FOREIGN KEY (`project`) REFERENCES `project` (`id`) )
ENGINE=InnoDB;
```

In this table 'timetrack', we have defined 2 foreign keys: one to the user, and one to the project. We also defined a date and a start and end time.
To add relation (foreign key) we can use also this sql:

```
ALTER TABLE `timetrack`
  ADD CONSTRAINT `fk_project_project` FOREIGN KEY (`project`) REFERENCES `project` (`id`),
  ADD CONSTRAINT `fk_user_user` FOREIGN KEY (`user`) REFERENCES `user` (`id`);
```

CODE GENERATION – STEP 2

Go to your main page of QCubed and click the codegen link. You should see something like the following screen:



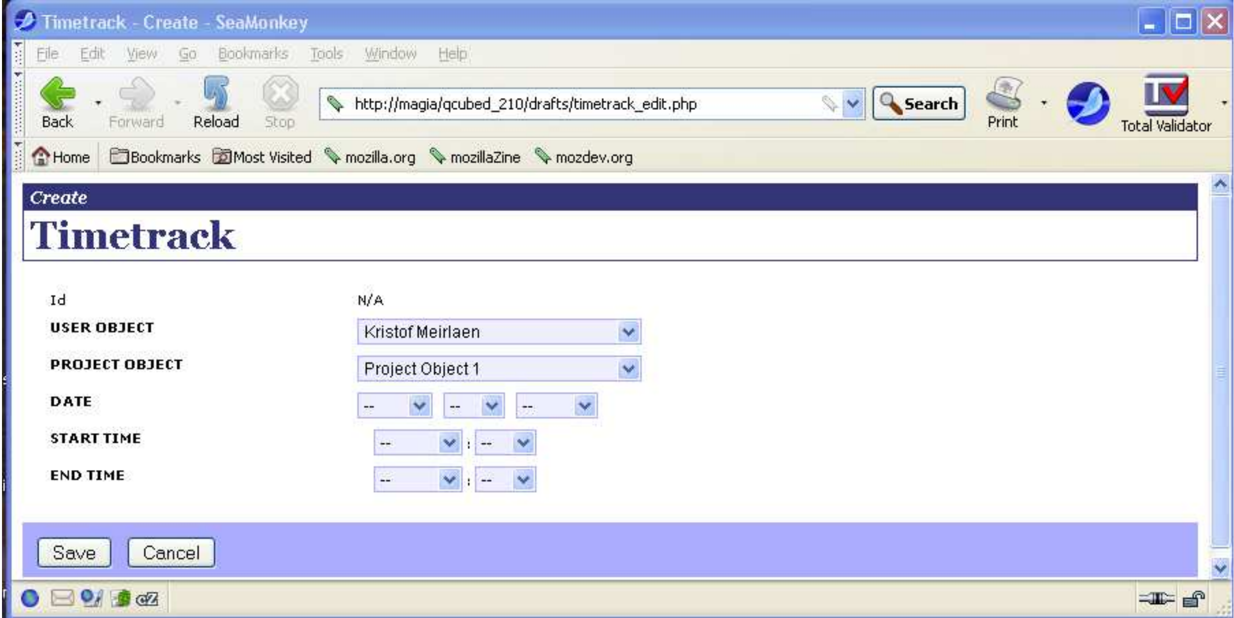
As you can see, in addition to the new Timetrack table, QCubed also re-code the generated Project and User tables.

LOOK AND FEEL PERSONALIZATION.

To apply personal preferred graphical layout (document size, color, font, controls etc.).as used in following screen we must copy my preferred `styles.css` in `\assests_core\css` and remove some lines in `header.inc` and `footer.inc.` My look and feel files are listed the the end of this chapter.

ADDING TIME TRACKS

As we would expect, in addition to the Data Objects for the Timetrack table, QCubed also created a form draft to create and view the data in the Timetrack table, so let us add some tracking:



As you can see, the User Object is already correctly filled since we adjusted previously to return the firstname and lastname, but now the Project object shows the "Project Object 1". Again, this is because QCubed does not know what to display. You will have to change the default return string of the Project object the same way as you did for the User Object. Go ahead and do it (Qustion: :applay patch to ?   Replay : includes\model\Project.class).

```
return sprintf('%s',  $this->strName);
```

After making the change, refresh the "create Timetrack" page. The project object should now display the project name instead of "Project Object".

CHANGING DEFAULT VALUES

The default Date, Starttime and End time values are a bit user unfriendly at this time. Every time we want to add a record, we need to modify the date, start time and end time to match the current date (we assume that we will usually log our time on the day we have performed it) and time. It would be better if the default date and start/endtime would be the current date and time.

REUSE AVAILABLE CODE

Before we do interventions on generated code, I introduce some trick to make our code modifications independent from generated code and also put the stone for create an application with minimal intervention on the available code, retaining, at the same time, the drafts form and dashboard panel for basic interaction whit our database table.

The page to create and edit the record is the timetrack_edit.php. As we know, QCubed generates 4 files for this form:

- `drafts\timetrack_edit.php`
- `drafts\timetrack_edit.tpl.php`
- `drafts\timetrack_list.php`
- `drafts\timetrack_list.tpl.php`

and 4 files for the panels:

- `drafts\panels\TimetrackEditPanel.class.php`
- `drafts\panels\TimetrackListPanel.class.php`
- `drafts\panels\TimetrackEditPanel.tpl.php`
- `drafts\panels\TimetrackListPanel.tpl.php`

This 8 files are in a generated folder, which means these will be overwritten if we would re-code generate.

For every table I suggest to create a subdir (the name can be the same of the table) and copy there the draft and drafts\panels generated script for the table. With some simple modification to a copied `form_drafts.php` from `\assets_core\php_devtools\` or `panel_drafts.php` and `panel_drafts.tpl.php` from `\assets_core\php_devtools\` we become able to use the copy of the generated forms and panels located in subdir (our application) that will be untouched by a regeneration activity.

TO USE FORM

Modification to `form_drafts.php` to activate form copied from drafts

```
From:
….
$objDirectory = opendir(__DOCROOT__ . __FORM_DRAFTS__);
….
foreach ($strObjectArray as $strObject=>$blnValue) {
  printf('<h1>%s</h1><p class="create"><a href="%s/%s_list.php">%s</a>
 |  <a href="%s/%s_edit.php">%s</a></p>',
$strObject, __VIRTUAL_DIRECTORY__ . __FORM_DRAFTS__, $strObject,
QApplication::Translate('View List'),
__VIRTUAL_DIRECTORY__ . __FORM_DRAFTS__, $strObject,
 QApplication::Translate('Create New'));
}
………
```

to:

```
…..
$objDirectory = opendir(dirname(__FILE__));
….
foreach ($strObjectArray as $strObject=>$blnValue) {
printf('<h1>%s</h1><p class="create"><a href="%s_list.php">%s</a>
 |  <a href="%s_edit.php">%s</a></p>',
$strObject,$strObject, QApplication::Translate('View List'),
$strObject, QApplication::Translate('Create New'));
}
…….
```

To instruct xxxxx_edit.php and xxxxxx_list.php to remain in this directory change also the copied timetrack_list.tpl.php

from:

```
…..
<h2 id="right"><a href="<?php _p(__VIRTUAL_DIRECTORY__ . __FORM_DRAFTS__) ?>/index.php">&laquo;
<?php _t('Go to "Form Drafts"'); ?></a></h2>
….
```

to:

```
…..
<h2 id="right"><a href="./form_drafts.php">&laquo; <?php _t('Go to "Form menu"'); ?></a></h2>
….
```

and from:

```
……..
<p class="create">
<a href="<?php _p(__VIRTUAL_DIRECTORY__ . __FORM_DRAFTS__) >/timetrack_edit.php">
<?php _t('Create a New'); ?><?php _t('Timetrack');?></a>
</p>
………
```

to (obviously *timetrack* can be different as it is the name of table to be edited) substitute only `<?php _p(__VIRTUAL_DIRECTORY__ . __FORM_DRAFTS__) .>` with `.` (dot):

```
……
<p class="create">
<a href="./timetrack_edit.php"><?php _t('Create a New'); ?>
<?php _t('Timetrack');?></a>
</p>
…….
```

Now with decision introduced in version 2.x to split the table generated forms in an abstract formbaseclass posted in includes\formbase_classes_generated and extended working form posted in \drafts a problem arise. To solve problem introduced in the constructed edit item link addess in timetrack_list.php (or other xxxx_list.php where xxxx is our table name) we can copy timetrack_edit.php from \drafts and function Form_Create() with the same function present in includes\formbase_classes_generated\TimetrackListFormBase.class.php and change:

from:

```
$strEditPageUrl = __VIRTUAL_DIRECTORY__ . __FORM_DRAFTS__ . '/timetrack_edit.php';
```

to:

```
$strEditPageUrl = './timetrack_edit.php';
```

or, if we want retain the capability of regen to follows db modification, the only solution I found at moment is copy in our \timetrack (or \xxxxxx whre xxxxxx is table name) the generated \drafts\timetrack_list.php (or \drafts\xxxxxxx_list.php) and override here (destroy and recreate column 0) what is done in `includes\formbase_classes_generated\TimetrackListFormBase.class.php`

whit this code (derived from `MetaAddEditLinkColumn` function present in generated `xxxxxDataGridGen.class.php`):

```
// modification  by magia 2012 as suggested in qcubed-quick-start-guide_v210_Chapt_05 step 2
// we retain all generated form control
// and insert here only the  modified
     protected function Form_Create() {
        parent::Form_Create();

// to use the local timetrack_edit.php and not that one present in drafts we must modify link
// located in column 0
        $strEditPageUrl = './timetrack_edit.php';
        $this->dtgTimetracks->RemoveColumn(0);
        $strColumnTitle = 'Edit';
        $strLinkHtml = 'Edit' ;
        $strEditPageUrl .= '/<?=urlencode($_ITEM->Id)?>';
        $strHtml = '<a href="' . $strEditPageUrl . '">' . $strLinkHtml . '</a>';
        $colEditColumn = new QDataGridColumn($strColumnTitle, $strHtml, 'HtmlEntities=False');
        $this->dtgTimetracks->AddColumnAt(0, $colEditColumn);
     }
```

And the last trick to have all ok for our table related subdir structure, in copied file `timetrack_edit.php` we must override the redirect function; so we can put inside `class TimetrackEditForm extends TimetrackEditFormBase { }` the appropriate redirect function (rememeber to substitute address to xxxx_list with the actual used):

```
  // Other Methods
     protected function RedirectToListPage() {
        QApplication::Redirect(__SUBDIRECTORY__.'/timetrack/timetrack_list.php');
     }
```

I suggest now, as finishing touch, to create in subdir a new small `index.php` with redirection to `form.drafts.php`:

```
<?php
header("Location: ./form_drafts.php");
?>
```

To use Panel

Using the Panel copied from dashboard is more easy and require only a one time modification. This is the solution I use in my developed application with a special, very easy redesigned `panel_drafts.php` and `panel_drafts.tpl.php` that skip the selection panel and display immediately list panel.
We will use my simple derived `panel_drafts.php` and`panel_drafts.tpl.php` that can be reused without any modification for all the subdir (remember one subdir per table).
`panel_drafts.php`
from:

```
$objDirectory = opendir(__DOCROOT__ . __PANEL_DRAFTS__);
…
     require(__DOCROOT__ . __PANEL_DRAFTS__ . '/' . $strClassName . 'ListPanel.class.php');
     require(__DOCROOT__ . __PANEL_DRAFTS__ . '/' . $strClassName . 'EditPanel.class.php');
```

to:

```
$objDirectory = opendir(dirname(__FILE__));
…
     require( $strClassName . 'ListPanel.class.php');
     require( $strClassName . 'EditPanel.class.php');
```

some code block are commented and some added as follows:

```
/* magia 2009 comment out
        $this->lstClassNames = new QListBox($this);
        $this->lstClassNames->AddItem('- Select One -', null);

        // Use the strClassNameArray as magically determined above to aggregate the listbox of
classes
        // Obviously, this should be modified if you want to make a custom dashboard
        global $strClassNameArray;
        foreach ($strClassNameArray as $strKey => $strValue)
            $this->lstClassNames->AddItem($strKey, $strValue);
        $this->lstClassNames->AddAction(new QChangeEvent(), new QAjaxAction('lstClassNames_Change'));

        $this->objDefaultWaitIcon = new QWaitIcon($this);
end magia 2009 */
//add
/* define list panel */

$this->pnlList->RemoveChildControls(true);
        $this->pnlEdit->RemoveChildControls(true);
        $this->pnlEdit->Visible = false;

/* call to output list panel */

    global $strClassNameArray;
    global $strClassName;

//  $this->lstClassNames_Change();
$this->pnlList->RemoveChildControls(true);
    $this->pnlEdit->RemoveChildControls(true);
    $this->pnlEdit->Visible = false;

    this->lblTitle->Text = $strClassName;
$objNewPanel = new $strClassNameArray[$strClassName]($this->pnlList, 'SetEditPane', 'CloseEditPane');
//end add
…
/* commented out by magia 2009
    protected function lstClassNames_Change($strFormId, $strControlId, $strParameter) {
        // Get rid of all child controls for list and edit panels
        $this->pnlList->RemoveChildControls(true);
        $this->pnlEdit->RemoveChildControls(true);
        $this->pnlEdit->Visible = false;

        if ($strClassName = $this->lstClassNames->SelectedValue) {
            // We've selected a Class Name
            $objNewPanel = new $strClassName($this->pnlList, 'SetEditPane', 'CloseEditPane');
            $this->lblTitle->Text = $this->lstClassNames->SelectedName;
        } else {
            $this->lblTitle->Text = 'AJAX Dashboard';
        }
    }

end of commented out */
……..
```

panel_drafts.tpl.php will modified in:

```
<?php $strPageTitle= basename (dirname($_SERVER['PHP_SELF']),"/")." list Panel"?>
<?php require(__CONFIGURATION__ . '/header.inc.php'); ?>
<?php $this->RenderBegin(); ?>
  <div id="titleBar">
  <h2 id="right"><a href="./form_drafts.php">&laquo; Go to "<?php _p($strPageTitle);?>"</a></h2>
  <h2> </h2>
  <h1><?php $this->lblTitle->Render(); ?></h1>
</div>
  <div id="right">
        <?php $this->pnlEdit->Render(); ?>
        <?php $this->pnlList->Render(); ?>
  </div>
  <br clear="all" style="clear:both" />
<?php $this->RenderEnd(); ?>
<?php require(__CONFIGURATION__ . '/footer.inc.php'); ?>
```

Note: only a simple modification is required on panels generated files:in TimetrackEditPanel.class.php and TimetrackListPanel.class.php we must change line related to template location:

```
// Setup Callback and Template
$this->strTemplate = './TimetrackEditPanel.tpl.php';
```

```
// Setup the Template
$this->Template = './TimetrackListPanel.tpl.php';
```

This is directory content for timetrack (**Forms version**):

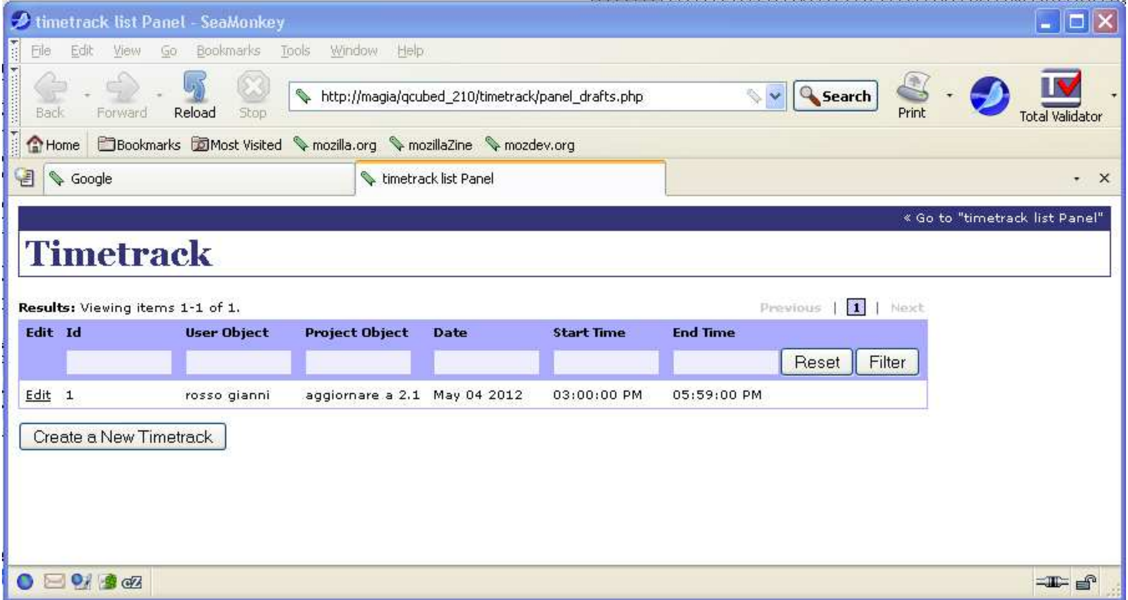
```
Directory of D:\wamp\www\qcubed_210\timetrack
02/05/2012  18.02           1.688 form_drafts.php
03/05/2012  10.56              48 index.php
03/05/2012  14.55           1.772 timetrack_edit.php
02/05/2012  17.32           1.270 timetrack_edit.tpl.php
03/05/2012  15.12           2.264 timetrack_list.php
03/05/2012  11.02             954 timetrack_list.tpl.php
```

Now if you go to page http://magia/qcubed_210/timetrack/form_drafts.php you can see:

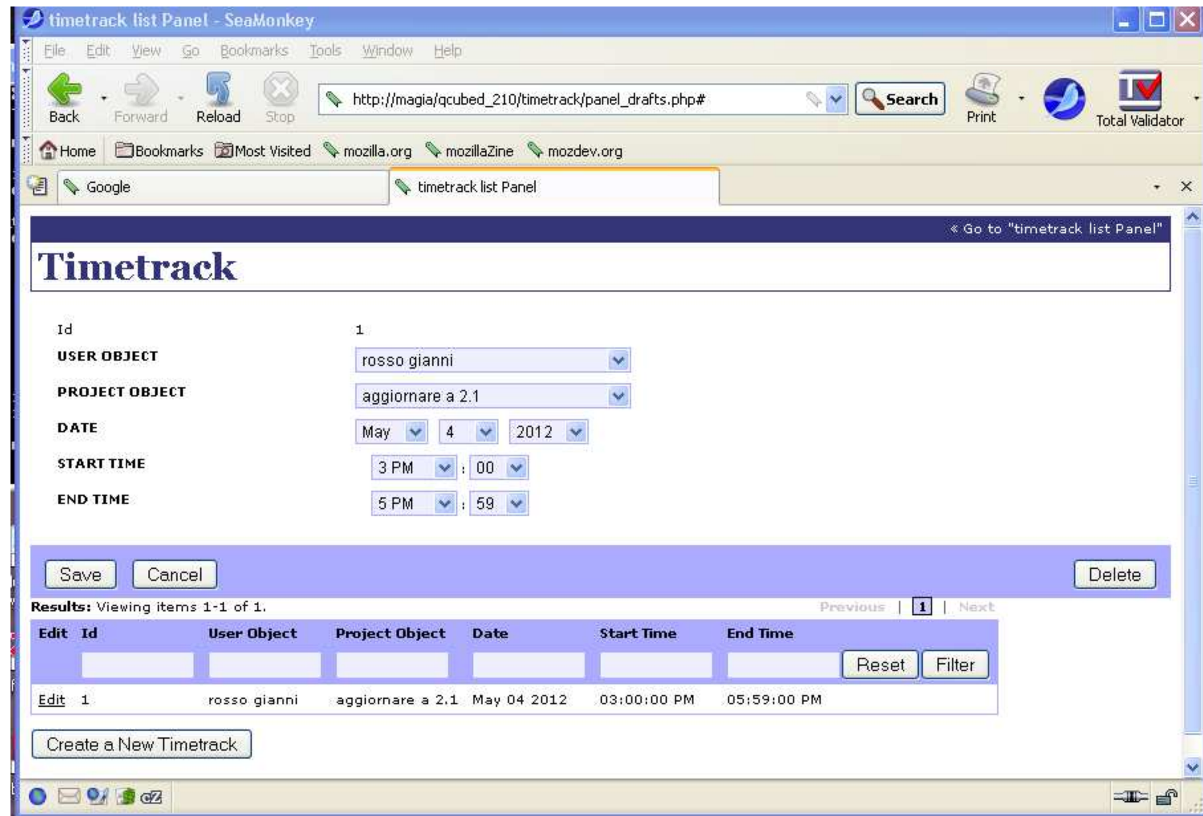
This is directory content for timetrack (**Panel version**):

```
Directory of D:\wamp\www\qcubed_210\timetrack
20/12/2010  23.39              49 index.php
21/12/2010  00.17           4.990 panel_drafts.php
03/05/2012  15.58             604 panel_drafts.tpl.php
03/05/2012  16.13           4.427 TimetrackEditPanel.class.php
02/05/2012  17.32             904 TimetrackEditPanel.tpl.php
03/05/2012  16.13           4.041 TimetrackListPanel.class.php
02/05/2012  17.32             419 TimetrackListPanel.tpl.php
```

Or you can go to page http://magia/qcubed_210/timetrack

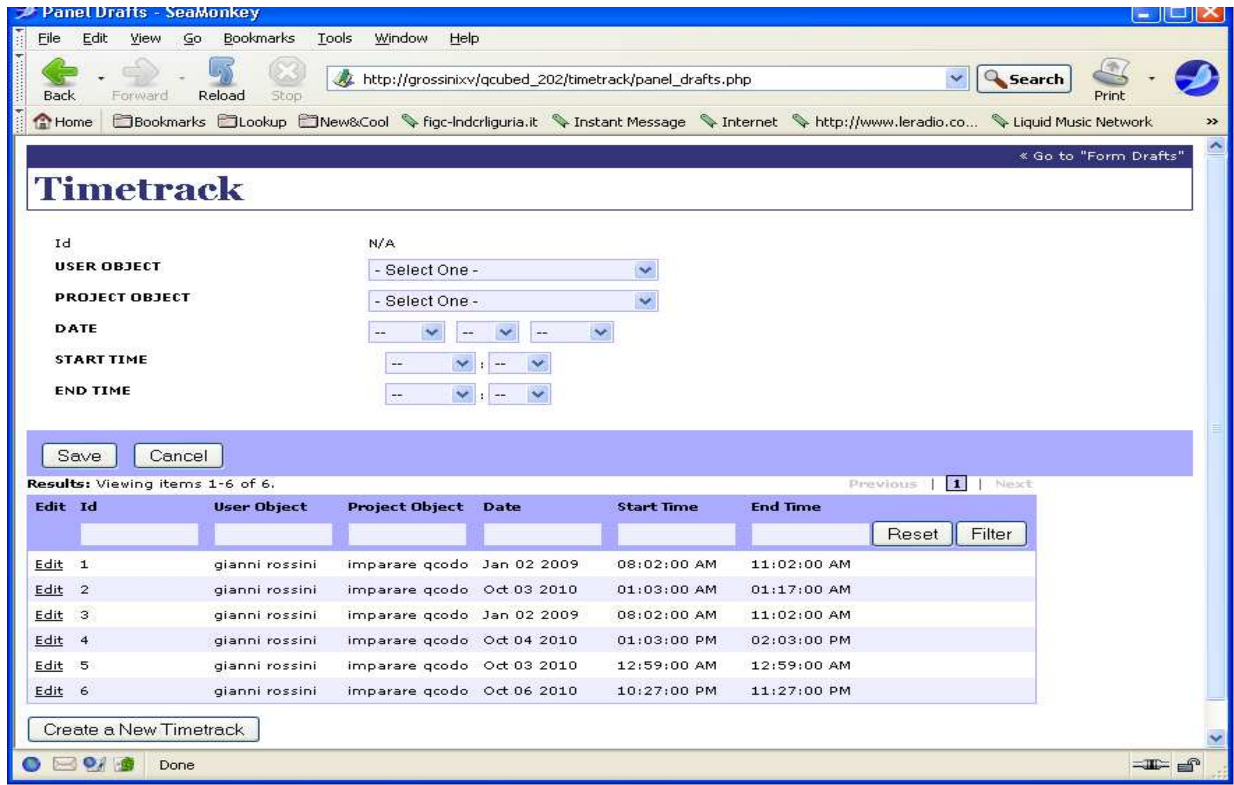


Choose edit 1 and you will see:



NOW WE CAN RETURN TO INTERVENTION REQUIRED TO CHANGE DEFAULT VALUE.

Remember what we told before: The default Date, Starttime and End time values are a bit user unfriendly at this time. Every time we want to add a record, we need to modify the date, start time and end time to match the current date (we assume that we will usually log our time on the day we have performed it) and time. It would be better if the default date and start/endtime would be the current date and time.



We can apply our *change default value code* to our version of `timetrack_edit.php` or `TimetrackEditPanel.class.php` copied in `\timetrack` or apply our clever modifications to field start/endtime in `includes\meta_controls\TimetrackMetaControl.class.php` so this mods is available always, and survives to regeneration and is present also in drafts.

So we must copy from `includes\meta_controls\generated\TimetrackMetaControlGen.class.php` those metacontrol creation functions:

```
function calDate_Create(){ ….}

function calStartTime_Create(){ ….}

function calEndTime_Create(){ …. }
```

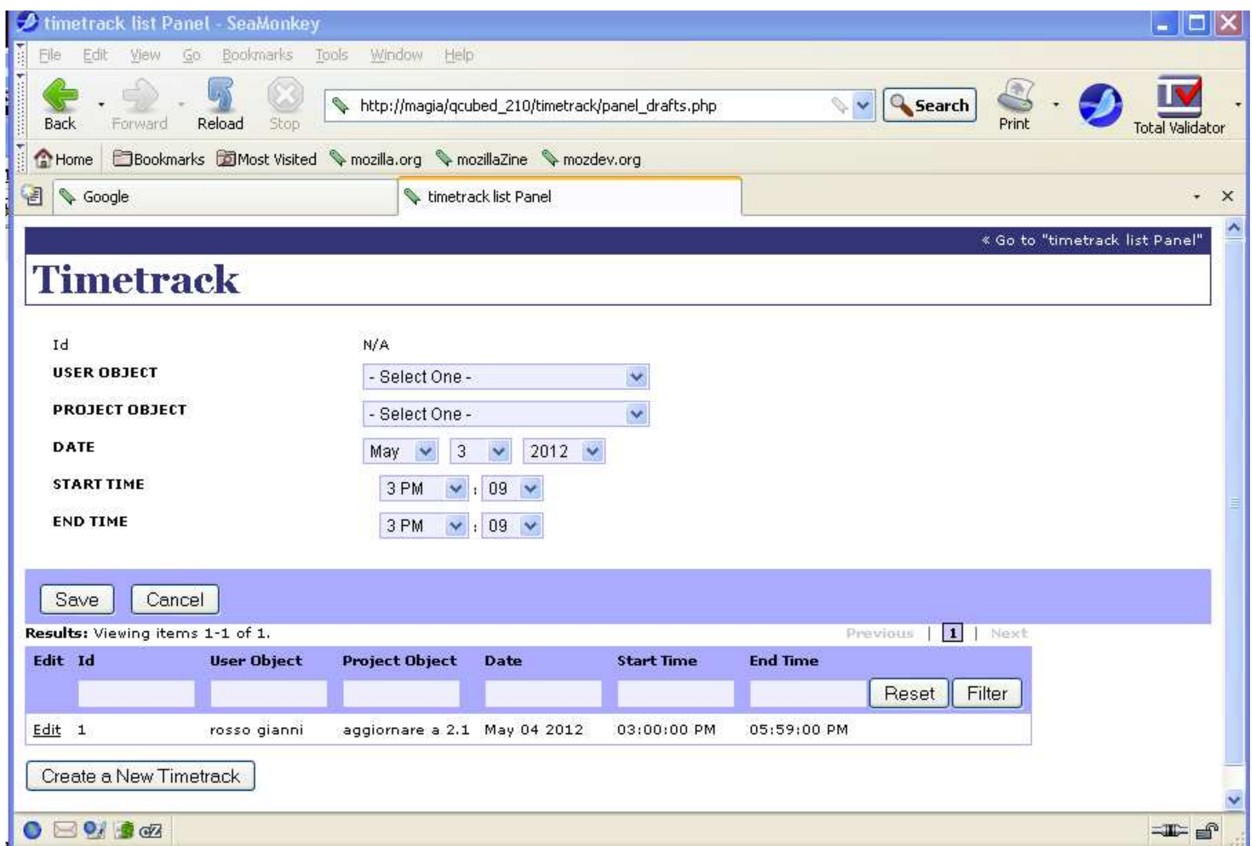
and apply the clever modification inside them.

Note: `blnEditMode` is a private variable indication if a form is used to Edit (true) or Create (false) a data entry.

includes\meta_controls\TimetrackMetaControl.class.php

```
<?php
  require(__META_CONTROLS_GEN__ . '/TimetrackMetaControlGen.class.php');
  /* This is a MetaControl customizable subclass, providing a QForm or QPanel access to event handlers
   * and QControls to perform the Create, Edit, and Delete functionality of the
   * Timetrack class.  This code-generated class extends from
   * the generated MetaControl class, which contains all the basic elements to help a QPanel or QForm
   * display an HTML form that can manipulate a single Timetrack object.
   * To take advantage of some (or all) of these control objects, you
   * must create a new QForm or QPanel which instantiates a TimetrackMetaControl class.
   * This file is intended to be modified.  Subsequent code regenerations will NOT modify
   * or overwrite this file.
   * @package My QCubed Application
   * @subpackage MetaControls      */
  class TimetrackMetaControl extends TimetrackMetaControlGen {
      // Initialize fields with default values from database definition
/*
      public function __construct($objParentObject, Timetrack $objTimetrack) {
         parent::__construct($objParentObject,$objTimetrack);
         if ( !$this->blnEditMode ){
            $this->objTimetrack->Initialize();
         }
      }
*/ // override generated metacontrols
      /* Create and setup QDateTimePicker calDate
       * @param string $strControlId optional ControlId to use
       * @return QDateTimePicker       */
      public function calDate_Create($strControlId = null) {
         $this->calDate = new QDateTimePicker($this->objParentObject, $strControlId);
         $this->calDate->Name = QApplication::Translate('Date');
         if  (!$this->blnEditMode)  {
            $this->calDate->DateTime = new QDateTime(QDateTime::Now);
      } else {
         $this->calDate->DateTime = $this->objTimetrack->Date;
      }
         $this->calDate->DateTimePickerType = QDateTimePickerType::Date;
         $this->calDate->Required = true;
         return $this->calDate;
      }
      /*  Create and setup QDateTimePicker calStartTime
       * @param string $strControlId optional ControlId to use
       * @return QDateTimePicker       */
      public function calStartTime_Create($strControlId = null) {
         $this->calStartTime = new QDateTimePicker($this->objParentObject, $strControlId);
         $this->calStartTime->Name = QApplication::Translate('Start Time');
         if  (!$this->blnEditMode)  {
            $this->calStartTime->DateTime = new QDateTime(QDateTime::Now);
      } else {
            $this->calStartTime->DateTime = $this->objTimetrack->StartTime;
      }
         $this->calStartTime->DateTimePickerType = QDateTimePickerType::Time;
         $this->calStartTime->Required = true;
         return $this->calStartTime;
      }
      /* Create and setup QDateTimePicker calEndTime
       * @param string $strControlId optional ControlId to use
       * @return QdateTimePicker       */
      public function calEndTime_Create($strControlId = null) {
         $this->calEndTime = new QDateTimePicker($this->objParentObject, $strControlId);
         $this->calEndTime->Name = QApplication::Translate('End Time');
         if  (!$this->blnEditMode)  {
               $this->calEndTime->DateTime = new QDateTime(QDateTime::Now);
         }else {
               $this->calEndTime->DateTime = $this->objTimetrack->EndTime;
         }
         $this->calEndTime->DateTimePickerType = QDateTimePickerType::Time;
         $this->calEndTime->Required = true;
         return $this->calEndTime;
      }
  }
?>
```

recall creation page and this is the result :



Note If time in fields is incorrect (shifted by one or two hours) you can go to configuration inc.php or better to php.ini to adjust it:

timezone setting in confguration.in.php

```
//date_default_timezone_set('America/Los_Angeles');
date_default_timezone_set('Europe/Rome');
```

If this change appear whit no effect, to let timezone setting in configuration inc able to work please cooment out this line in php.ini:

```
;date.timezone = UTC
```

ADDING VALIDATION IN FORMS

By default, the code generated by QCubed will already perform some basic validation based on the database model: if a field is marked as "not null", the generated code will implement checks so that the field is correctly filled with data before storing it to the database. In case a field is found empty or invalid (e.g. The database field type is integer, but the entered value is text), a warning is displayed explaining the reason as to why the validation failed.

In our application, did you notice that one could enter an end time which is lower then the start time? Of course, this would result in some negative time. Therefore, we will implement a validation for this.

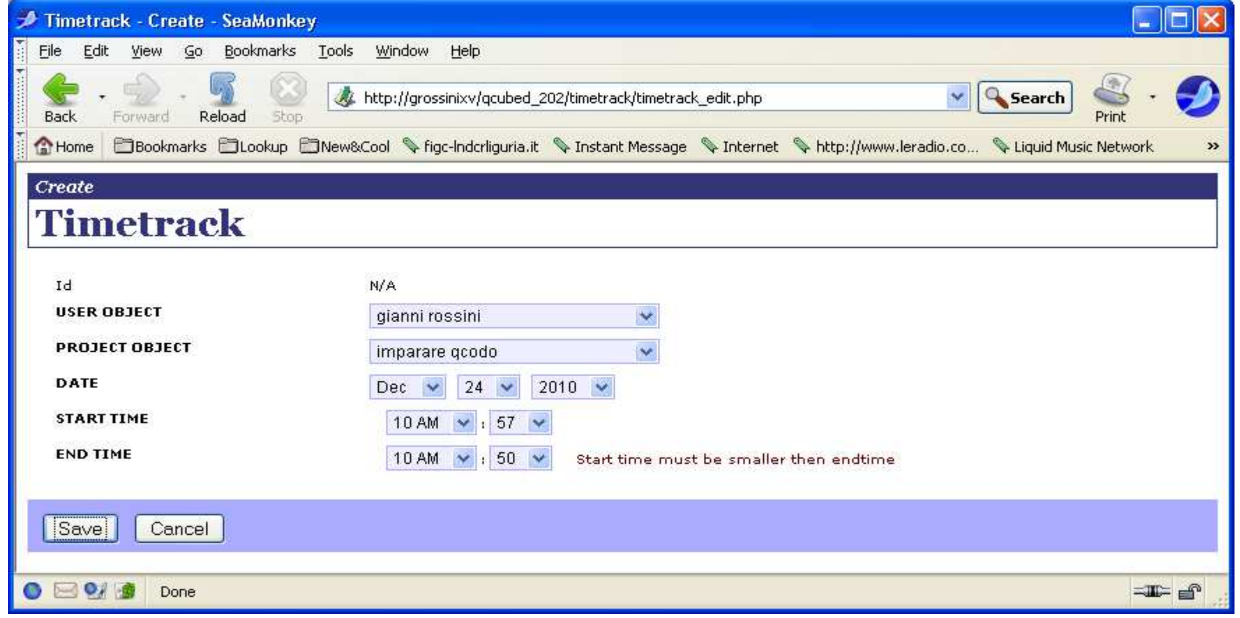
Using QCubed, the best way to add validation to a form is to put a special function into the object form class file called "Form_Validate()". This function is automatically called by QCubed whenever the "Save" button is clicked

(Note: in fact, the Save button has a property name "CausesValidation" set to "true" when it was created, causing the Form_Validate() to be called when the button is clicked)

Put this code in our timetrack_edit.php file inside function Form_Validate() copied from includes\formbase_classes_generated\TimetrackEditFormBase.class.php:

```
protected function Form_Validate() {
        // By default, we report that Custom Validations passed
        $blnToReturn = true;
// modification  by magia 2012 as suggested in qcubed-quick-start-guide_v210_Chapt_05 validation
        // Custom Validation Rules
        // TODO: Be sure to set $blnToReturn to false if any custom validation fails!

        if ($this->calStartTime->DateTime > $this->calEndTime->DateTime){
           $this->calEndTime->Warning = "Start time must be smaller then endtime";
           $blnToReturn =  false;
        }

        $blnFocused = false;
        foreach ($this->GetErrorControls() as $objControl) {
           // Set Focus to the top-most invalid control
           if (!$blnFocused) {
              $objControl->Focus();
              $blnFocused = true;
           }

           // Blink on ALL invalid controls
           $objControl->Blink();
        }

        return $blnToReturn;
     }
```

Now, when the btnSave is clicked, this function will be executed. If the Start Time is greater then the End Time, we display an error on the End Time and return the form.

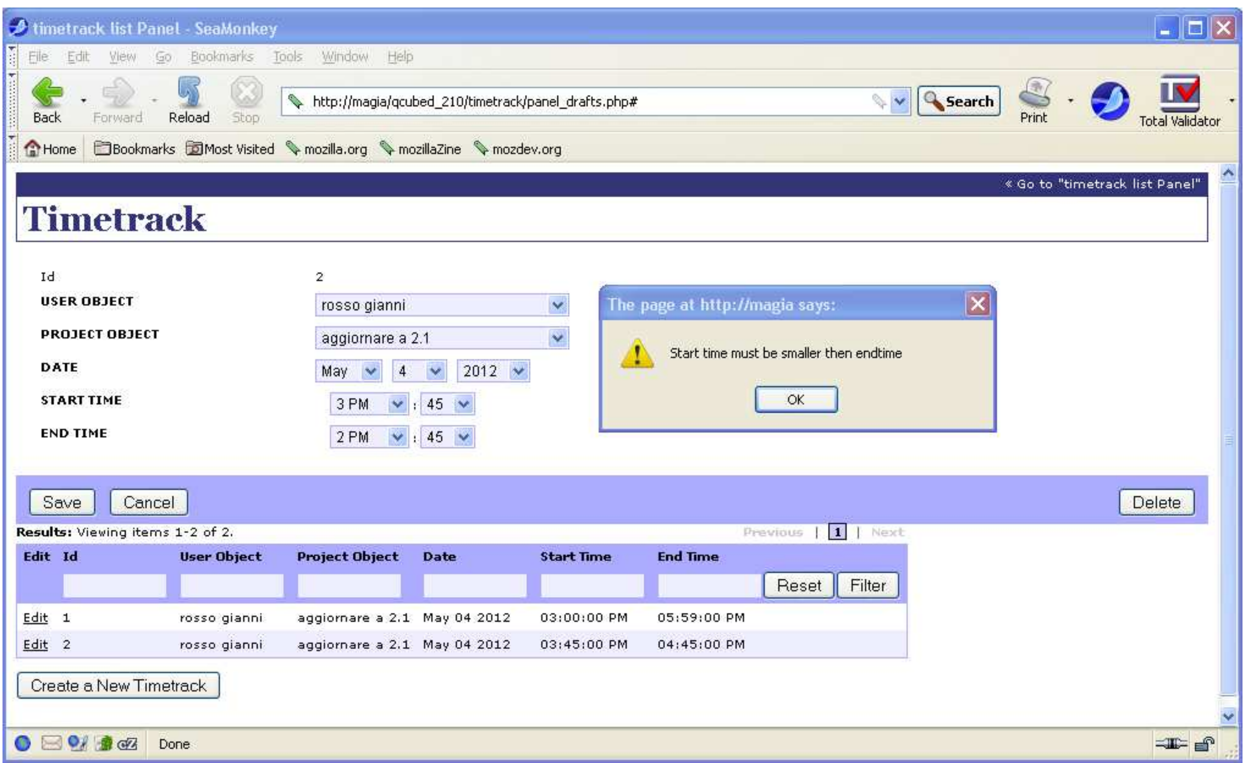


ADDING VALIDATION IN PANELS

To achive Validation in Panels we can do a small different operation on our \timetrack\TimetrackEditPanel.class.php, adding Validate{} function .

```
// modification  by magia 2012 as suggested in qcubed-quick-start-guide_v210_Chapt_05 panel validation
    public function Validate() {
        if ($this->calStartTime->DateTime > $this->calEndTime->DateTime){
            QApplication::DisplayAlert("Start time must be smaller then endtime");
            $this->calStartTime->Focus();
            return false;
        }
        return true;
    }
```

Now, when the btnSave is clicked, this function will be executed. If the Start Time is greater then the End Time, we display an error on the End Time and return the Panel.



At the end of chapter we choose to put in table related directory all form and panel whit standerd index.php pointing to CRUD Panels and index_forms.php pointing to CRUD Forms for the table.

SUMMARY

In this chapter you have learned to extend the behavior of the forms that QCubed generated by default. If you understand how to extend and build upon the form drafts, continue to the next chapter, where we will create a more advanced application.

```
As promised, my styles.css list  ( location is \assets\_core\css\)

  /* These are all EXAMPLES -- they are meant to be updated/changed/modified */
          body { font: 10px 'Verdana', 'Arial', 'Helvetica'; }

  /* Standard HTML Form Controls */
   input.button { border: 1px solid #aaf; font: 11px 'Arial', 'Helvetica'; font-weight: bold;
  background-color: #eef;}
   input.button:hover { background-color: #fff; }

   input.textbox, textarea.textbox { font: 12px 'Arial', 'Helvetica'; background-color: #eef;
  border: 1px solid #aaf; padding: 2px 3px 2px 3px;}
   textarea.textbox { height: 120px; }
     input.textbox:focus, textarea.textbox:focus {background-color: #fff; border-color: #aaf;}

   select.listbox { font: 12px 'Arial', 'Helvetica'; width: 208px; background-color: #eef; border:
  1px solid #aaf; }
     select.listbox:focus { background-color: #fff; border-color: #aaf;}
     a.listboxReset { font-family: verdana, arial, helvetica; font-size: 8pt; text-decoration:
  none; color: #337; }
     a.listboxReset:hover { text-decoration: underline; }

  /**
   * Various QControl-specific Styles
  */

  /* QFileAsset */
   div.fileassetDbox { border: 4px solid #333; background-color: #fff; padding: 20px; width:
  400px; overflow: auto; }
     div.fileassetDbox h1 { margin: 0; }
     div.fileassetDbox input { padding: 3px; margin-right: 5px; }

  /* QDialogBox */
   div.dialogbox { border: 4px solid #333; background-color: #fff; padding: 20px; width: 400px;
  overflow: auto; }

  /* QDateTimePicker */
   span.datetimepicker { }
     span.datetimepicker select { font: 12px 'Arial', 'Helvetica'; background-color: #eef; border:
  1px solid #aaf; }
     span.datetimepicker select:focus { background-color: #fff; border-color: #aaf; }
     span.datetimepicker select.month { width: 55px; }
     span.datetimepicker select.day { width: 45px; margin-left: 8px; }
     span.datetimepicker select.year { width: 60px; margin-left: 8px; }
     span.datetimepicker select.hour { width: 65px; margin-left: 12px; margin-right: 2px; }
     span.datetimepicker select.minute { width: 45px; margin-left: 2px; margin-right: 2px; }
     span.datetimepicker select.second { width: 45px; margin-left: 2px; }
```

```
/* QDataGrid */
 table.datagrid { border: 1px solid #aaf; }
    table.datagrid caption, table.datagrid tfoot { padding-bottom: 4px; overflow: auto; }
/*magia 2009 - to center pagination page_number*/
table.datagrid span.center { float: left; font-size: 10px; display: block; }
       table.datagrid span.left { float: left; font-size: 10px; display: block; }
       table.datagrid span.right { float: right; font-size: 10px; display: block; }
    table.datagrid tr { background-color: #fff; }
       table.datagrid tr.alternate { background-color: #eef; }
    table.datagrid th { background-color: #aaf; padding: 3px 5px 3px 5px; text-align: left; }
       table.datagrid th a { color: #000; text-decoration: none; }
       table.datagrid th a:hover { color: #000; text-decoration: underline;}
    table.datagrid td { padding: 5px; }
       table.datagrid td a { color: #000; }
       table.datagrid td a:hover { color: #000; text-decoration: none; }
    table.datagrid tfoot { border-top: 1px solid #aaf;}

/* QPaginator */
 span.paginator { margin: 0; padding: 0; }
 span.paginator span { list-style-type:none; display: inline; padding: 0; margin: 0; }
       span.paginator span.page a { text-decoration: none; color: #000000; padding: 0 3px 0 3px; 
margin: 0 2px 0 2px;}
       span.paginator span.page a:hover { background-color: #ccf; }
       span.paginator span.arrow { font-weight: bold; color: #ccc; margin: 0; padding: 0 3px 0 
3px; }
       span.paginator span.arrow a { font-weight: bold; color: #000; text-decoration: none; }
       span.paginator span.selected { font-weight: bold; background-color: #ccf; padding: 0 3px 0 
3px; margin: 0 2px 0 2px; border: 1px; border-style: solid; }
       span.paginator span.break { color: #666; margin: 0 5px 0 5px; }
       span.paginator span.ellipsis { color: #666; }

/* QCalendar */
 div.calendar { width: 200px; border: 1px solid #337; background-color: #ccf; }
    div.calendar div.navigator { background-color: #337; overflow: auto; height: 100%; color: 
#fff; padding: 2px 6px 2px 6px; }
    div.calendar div.navigator div.left { float: left; }
    div.calendar div.navigator div.month { float: left; width: 80px; text-align: center; font-
weight: bold; font-size: 11px; }
    div.calendar div.navigator div.year { float: right; }
    div.calendar div.navigator div.year span { font-size: 11px; font-weight: bold; margin-left: 
6px; margin-right: 6px; }
       div.calendar div.navigator a { color: #fff; text-decoration: none; }
       div.calendar div.navigator a:hover { text-decoration: underline; }
    div.calendar table { border: 0; margin-left: auto; margin-right: auto;}
       div.calendar th { text-align: center; border: 0; padding: 4px 2px 4px 2px; font-weight: 
bold; font-size: 10px;}
       div.calendar td { text-align: center; border-width: 0 0 1px 0; border-style: solid; 
border-color: black; padding: 0;}
          div.calendar td.lastRow { border-width: 0;}
          div.calendar td a { text-decoration: none; color: #000; display: block; padding: 2px 
6px 2px 6px; }
          div.calendar td.today { background-color: #bbd; }
          div.calendar td.nonMonth a { color: #aaf; }
          div.calendar td.selected { background-color: #c9a; font-weight: bold; }
          div.calendar td a:hover { background-color: #eef; color: #000; }
```

```
    div.calendar div.options { text-align: center; background-color: #337; }
       div.calendar div.options a { color: #fff; font-size: 10px; text-decoration: none;}
       div.calendar div.options a:hover { text-decoration: underline; }
/**
 * Styles for Custom Render Functions as defined in QControl.class.php
 */
 div.renderWithName { padding: 4px 0 4px 0; overflow: auto; height: 100%; }
    div.renderWithName div.left { float: left; width: 200px; margin-right: 25px; }
    div.renderWithName div.required { font-weight: bold; text-transform: uppercase; }
    div.renderWithName div.left span.instructions { font: 10px 'Verdana', 'Arial', 'Helvetica';
color: #999; font-style: italic; text-transform: none;}
    div.renderWithName div.right { }
    div.renderWithName div.right span.error { margin-left: 15px; font: 10px 'Verdana', 'Arial',
'Helvetica'; color: #600;}

/**
 * Form and Dashboard Draft-Specific Styles
 * NOTE: because these are page specific, we use element IDs instead of element classes for most
items here
 * (e.g. we will never have more than one titleBar per page)
 */
 div#titleBar { border: 1px solid #337; margin-bottom: 18px; }
    div#titleBar h2 { background-color: #337; font: 12px 'Georgia', 'Times New Roman', 'Times';
font-style: italic; font-weight: bold; color: #fff; margin: 0; padding: 2px 5px 2px 5px; float:
none; }
    div#titleBar h2#right { float: right; font: 10px 'Verdana', 'Arial', 'Helvetica'; font-style:
normal; font-weight: normal; }
    div#titleBar h2#right a { color: #fff; text-decoration: none; }
    div#titleBar h2#right a:hover { text-decoration: underline; }
    div#titleBar h1 { font: 28px 'Georgia', 'Times New Roman', 'Times'; color: #337; margin: 0;
padding: 2px 0 2px 5px; font-weight: bold; }

 p.create { }
    p.create a { text-decoration: none; color: #337; font-weight: bold; }
    p.create a:hover { text-decoration: underline; }

 div#draftList { }
    div#draftList h1 { margin: 0; font: 20px 'Georgia', 'Times New Roman', 'Times'; font-weight:
bold; }
    div#draftList p { margin-top: 6px; margin-bottom: 18px;}
 div#formControls { padding: 0 0 0 20px; }
 div#formActions { background-color: #aaf; padding: 10px; height: 20px; margin-top: 18px; }
    div#formActions div#save { float: left; }
    div#formActions div#cancel { margin-left: 10px; float: left; }
    div#formActions div#delete { float: right; }

 div#dashboard { }
    div#dashboard div#left { float: left; width: 200px; height: 100px; font-size: 10px;
background-color: #eef; border: 1px solid #000; padding: 5px; }
    div#dashboard div#right { float: left; margin-left: 10px; width: 760px; }
    div#dashboard div#right div#pnlList { margin-bottom: 10px; border: 1px solid #000; padding:
5px; height: 300px; overflow: auto; }
    div#dashboard div#right div#pnlEdit { border: 1px solid #000; padding: 5px; background-color:
#eef; height: 240px; overflow: auto; }
```

includes\configuration\header.inc

```
<?php
 // This example header.inc.php is intended to be modfied for your application.
?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN" "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
 <head>
    <meta http-equiv="Content-Type" content="text/html; charset=<?php
_p(QApplication::$EncodingType); ?>" />

<?php if (isset($strPageTitle)) { ?>      <title><?php _p($strPageTitle); ?></title>
<?php } ?>

<style type="text/css">@import url("<?php _p(__VIRTUAL_DIRECTORY__ .
__CSS_ASSETS__); ?>/styles.css");</style>
 </head>
 <body>
    <div id="page">

<?php /*  commented out
       <div id="header">
          <div id="headerLeft">
             <div id="codeVersion"><span class="headerSmall">QCubed Development Framework <?php
_p(QCUBED_VERSION) ?></span></div>
             <div id="pageName"><?php if (isset($strPageTitle)) { _p($strPageTitle); } ?></div>
          </div>
          <div id="headerRight">
          </div>
       </div>
*/ ?>
       <div id="content">
```

includes\configuration\footer.inc

```
<?php
 // This example footer.inc.php is intended to be modfied for your application.
?>
<?php
/* commented out
       </div>
       <div id="footer">
          <div id="footerLeft"><a href="http://qcu.be/"><img src="<?php _p(__VIRTUAL_DIRECTORY__ .
__IMAGE_ASSETS__  .  '/qcubed_logo_footer.png');  ?>"  alt="QCubed  -  A  Rapid  Prototyping  PHP5
Framework" /></a></div>
          <div id="footerRight">
             <div><span class="footerSmall">For more information, please visit the QCubed website
at <a href="http://www.qcu.be/" class="footerLink">http://www.qcu.be/</a></span></div>
             <div><span class="footerSmall">Questions, comments, or issues can be discussed at the
<a href="http://qcu.be/forum" class="footerLink">Examples Site Forum</a></span></div>
          </div>
       </div>
*/ ?>
    </div>
 </body>
</html>
```

CHAPTER 6: PUTTING IT TOGETHER: CREATING A QCUBED APPLICATION

An application usually is a little more then some basic forms: users have to identify themselves in order to perform time tracking, managers can view reports for their projects.

Now that we have the basic functionalities of a time tracking system, in this chapter, we will leverage this code base in order to create a complete time tracking application.

These are the requirements:

- users must be able to login and log time on projects they are assigned to (for now, they can do this on all projects)
- the passwords in the database are stored in a one way hash. This prevents unauthorized users from potentially viewing the password
- the user must be able to change his password
- the administrative tasks for creating projects and users is only accessible by a user called "admin". (as an alternative you could also make this a setting through the database. Do this as an exercise!)

CHANGING THE DATABASE MODEL

In order to accomplish our goals, we must adjust some parts of our database. Users should have a login name and a password assigned in order to login:

On our User table we add username column and his unique index (this is my case), and password field:

```
ALTER TABLE `user` ADD COLUMN `username` VARCHAR(50) NOT NULL DEFAULT '';
ALTER TABLE `user` ADD COLUMN `password` VARCHAR(50) NOT NULL DEFAULT '' AFTER `username`;
```

Populate new field `username` bef ore unique index creation on this field (DB rules requires that username data must be different and, of course, present before unique index creation) so use PHPmyAdmin to populate username field.

```
ALTER TABLE `user` ADD UNIQUE (`username`);
```

This adds uniquekey to the user table

Now can be time to regenerate. This causes QCubed to generate a User::LoadByUsername($strUsername) function, which returns a User object. We will use this function later in the process when we are creating the login forms.

You can also see that regen added a `function Validate()` in `\drafts\panels\UserEditPanel.class.php` and a `function Form_Validate()` in `includes\formbase_classes_generated\ UserEditFormBase.class.php`.

After regeneration, we could start assign the user a password. However, the password would be stored unencrypted. Also, to make sure that the administrator has entered the correct password, we need to extend our form so that it accepts 2 passwords. Let us implement the encryption and the extra validation field.

ADDING ENCRYPTION AND VALIDATION OF THE PASSWORD

Before doing so, let us re-code generate. Go back to the main QCubed page, and codegen the database. Note that nothing of the functionalities we implemented previously are lost.

ADDING AN EXTRA FIELD

To add an extra field, we will have to edit the form that allows us to add/edit the user. We will therefore extend the class UserEditForm to have this functionality. Remember that UserEditForm is a regular QForm class, so everything should be done just as you did in the introduction chapter on QForms.

As we did for timetrack we go in dir \user and copy there the user created file from `\drafts` (remember to add `function RedirectToListPage()`introduced in our \user version) and `\drafts\panels` (remember to modify `$this->strTemplate` as per suggestion in previous chapter).

First, add a new member in our `\user\user_edit.php`. Let us call it `$txtPassword2`.

```
 class UserEditForm extends UserEditFormBase {
......
  protected $txtPassword2;
......
```

Next, we will need to define what this member is, and how it should behave.

To do this, let us take a look on how it is done in
`includes\meta_controls\generated\UserMetaControlGen.class.php`!

All we need to do is copy the code used in `function txtPassword_Create`, paste it into `Form_Create` function and modify it a little so that it is handled for the `$txtPassword2` instead of `$txtPassword`.

In `UserMetaControl.class.php` we find the following code snippet:

```
public function txtPassword_Create($strControlId = null) {
        $this->txtPassword = new QTextBox($this->objParentObject, $strControlId);
        $this->txtPassword->Name = QApplication::Translate('Password');
        $this->txtPassword->Text = $this->objUser->Password;
        $this->txtPassword->Required = true;
        $this->txtPassword->MaxLength = User::PasswordMaxLength;
        return $this->txtPassword;
    }
```

Perfect! Let's copy the contents from this function to our form_create function that inherit all from the same function in includes\formbase_classes_generated\UserEditFormBase , and modify the member variable. Also adjust the name property to have the name "password2". The code should now look like this:

```
    protected function Form_Create() {
        parent::Form_Create();
        $this->txtPassword2 = new QTextBox($this);
        $this->txtPassword2->Name = QApplication::Translate('Password2');
        $this->txtPassword2->Text = $this->txtPassword->Text;
        $this->txtPassword2->Required = true;
        $this->txtPassword2->MaxLength = User::PasswordMaxLength;
}
```

Next is to draw the extra field on the form. Remember that a QForm has 2 files: one for the logic and one for the layout. In this case, these are the following 2 files:

- form_drafts\user_edit.php : the logic
- form_drafts\generated\user_edit.tpl.php : the layout

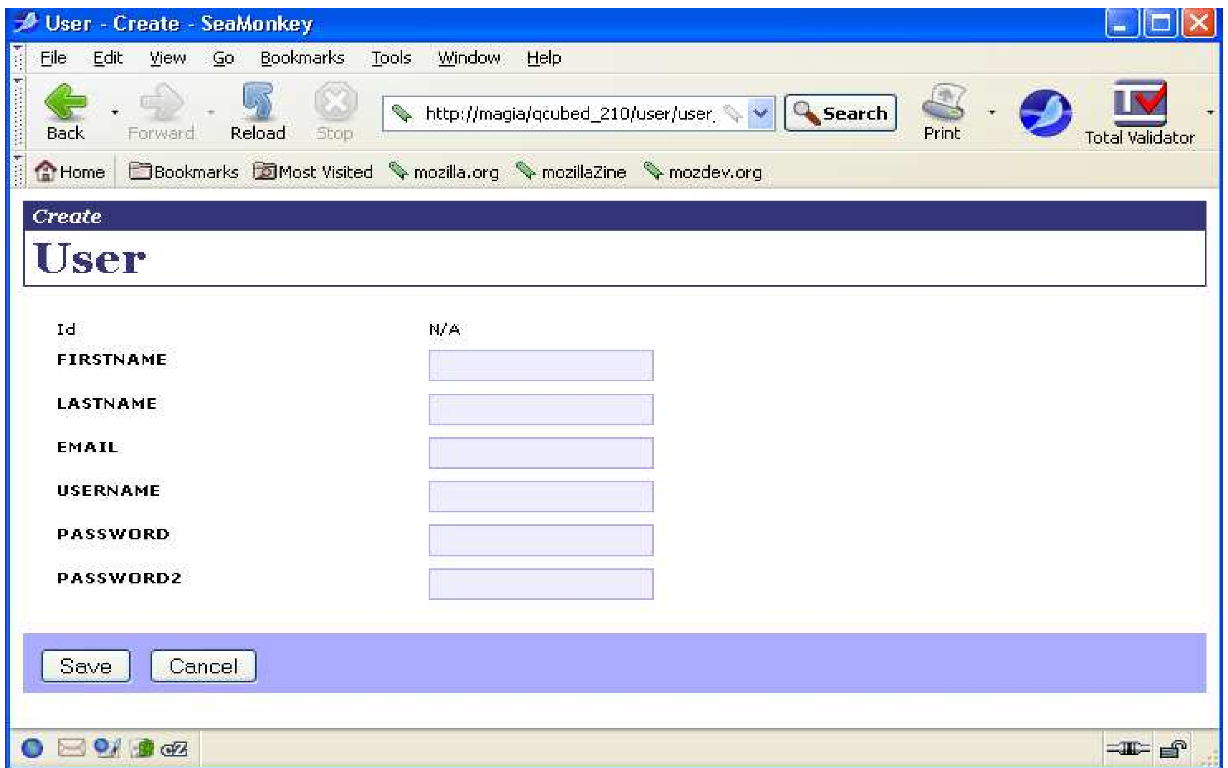
To draw the extra text box on the page, we should therefore modify `user_edit.tpl.php`.

We can edit the `user_edit.tpl.php` file. To include the extra text box: simply add the `RenderWithName` method of the `$txtPassword2` to the location where you want the text box to be drawn (you can copy/paste the code from the `txtPassword`):

```
...
<?php $this->txtPassword->RenderWithName();?>
<?php $this->txtPassword2->RenderWithName(); ?>
...
```

Remember to apply the correction (see what we did in chapt. 05 for timetrack) to `user_edit.php` , `user_list.php` and `user_list.tpl.php`, to avoid redirection or unwanted link to \drafts original code for user table basic CRUD function.

Save the file, and check the result by going to the `(magia is my localhost )` `http://magia/qcubed_102/user/user_edit.php` form in you browser:

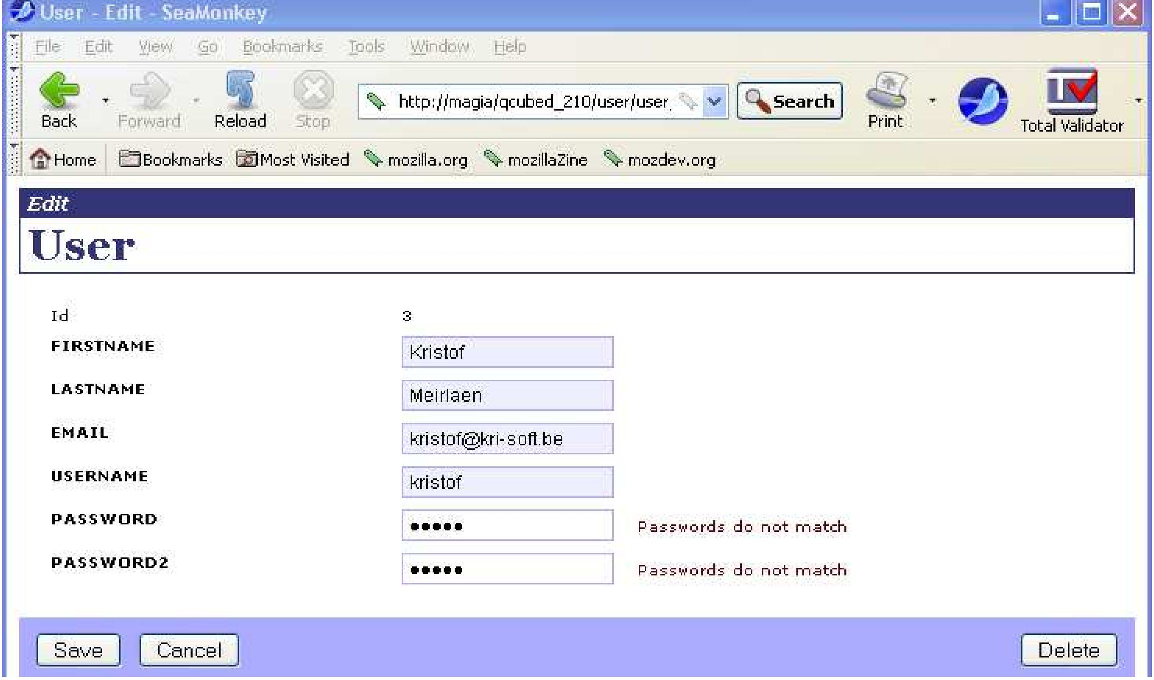


VALIDATION

Before saving the form, we need to check if the 2 passwords that were entered have the same value. We need to add a short validation to accomplish this, and just as we did in the previous example (but now in the user_edit form), we will create the "form_validate" function to verify if the passwords match:

```
protected function Form_Validate() {
// By default, we report that Custom Validations passed
    $blnToReturn = true;

    // Custom Validation Rules
    // TODO: Be sure to set $blnToReturn to false if any custom validation fails!
  if ($this->txtPassword2->Text !=  $this->txtPassword->Text)  {
    $this->txtPassword->Warning =  "Passwords  do  not  match";
    $this->txtPassword2->Warning =  "Passwords  do  not  match";
    $blnToReturn =  false;
  }
    $blnFocused = false;
    foreach ($this->GetErrorControls() as $objControl) {
    // Set Focus to the top-most invalid control
       if (!$blnFocused) {
          $objControl->Focus();
          $blnFocused = true;
       }
       // Blink on ALL invalid controls
       $objControl->Blink();
    }
    return $blnToReturn;
 }
```

Save the file, and check the result by going to the (magia is my localhost) `http://magia/qcubed_102/user/user_edit.php` form in you browser:



PASSWORD BOX – HOW WE HID THE PASSWORD

One final touch is not to display the password while it is typed. To do this, we know that in HTML, this is done by defining an <input type="password"> should be used. How is this handled in QCubed? We never had to put any <input> types in our code so far!

In QCubed everything is an object. The object of our password dialogs is now a QTextBox. In QTextBox, we can assign several properties for display such as color, css style, position, etc... In our case, we need to change the style to password.

QTextBox has a property called "TextMode". The TextMode property takes a QTextMode object as a valid value.

QTextMode has the following possible values:

- SingelLine (default for QtextBox)
- MultiLine
- Password

So to change the field from a normal text box to a password textbox, we need to set the TextMode property of the password fields to QTextBox::Password:

In our form_create, we added 2 lines:

```
$this->txtPassword2->Text  =  $this->objUser->Password;
$this->txtPassword2->Required  =  true;

$this->txtPassword2->TextMode = QTextMode::Password;
$this->txtPassword->TextMode = QTextMode::Password;

 }
```

So how we maked the passwords fields.

ENCRYPTION

Next requirement is to encrypt the password.

For the encryption of the password, we will use the one-way hashing function sha1. Before we store the value in the database, we make sure we perform the function sha1() on it.

The best way to do this in QCubed is to override the method SaveUser() inside the generated `UserMetaControlGen.class.php` . SaveUser() is a function inside the metacontrols that is created by QCubed and basically sets the values of the object to be saved to the values of the fields in the form. It is called whenever the "Save" button is pressed.

In the `UserMetaControlGen.class.php` file, this function looks like this:

```
   public function SaveUser() {
      try {
         // Update any fields for controls that have been created
         if ($this->txtFirstname) $this->objUser->Firstname = $this->txtFirstname->Text;
         if ($this->txtLastname) $this->objUser->Lastname = $this->txtLastname->Text;
            if ($this->txtEmail) $this->objUser->Email = $this->txtEmail->Text;
            if ($this->txtUsername) $this->objUser->Username = $this->txtUsername->Text;
            if ($this->txtPassword) $this->objUser->Password = $this->txtPassword->Text;
...
```

To override this function, copy the function inside the UserMetaControl.class.php, set the Password field to have a sha1() of the txtPassword form value and finally call the parent function to continue his job:

```
protected function SaveUser() {

if ($this->txtPassword) $this->txtPassword->Text =  sha1($this->txtPassword->Text);
 parent::SaveUser();

}
```

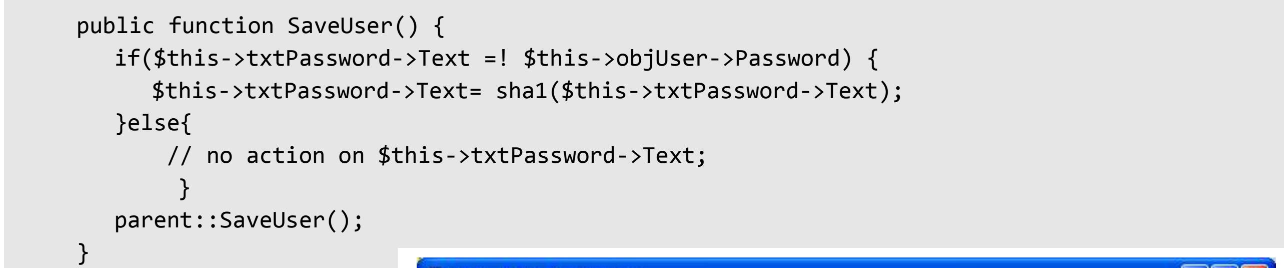
Testing it should show that it works. But it only works well when adding a user, or changing a password. In case we leave the password unchanged during an edit, the result of the stored value will be a hash of a hash, etc...

To avoid this, let us verify if the password was changed during the edit. If it was changed, we hash it, if it was not changed, we don't.

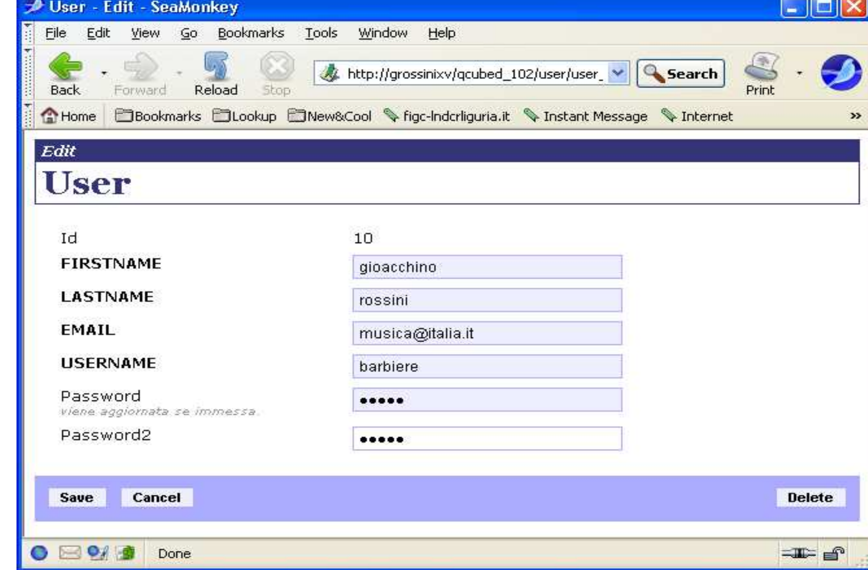
One way of doing this is to check the txtPassword field value against the previous value ($this->objUser->Password). If txtPassword field was changed, we know the user typed something, and we need to hash the field to save the new passord.

To do this we add to our `includes\meta_controls\UserMetaControl.class.php` in the function `SaveUser ()` a test on `txtPassord` input filed. If some data inserted and the data are different from original (crypted) data, we crypt `txtPassword`-data, otherwise we leave data untouched

After this test, we can leave control to parent generated `SaveUser()`function.



```
    public function SaveUser() {
       if($this->txtPassword->Text =! $this->objUser->Password) {
          $this->txtPassword->Text= sha1($this->txtPassword->Text);
       }else{
           // no action on $this->txtPassword->Text;
            }
       parent::SaveUser();
    }
```



new user creation

If you prefer use Panel, you can do same operation we did in chapter 05 for Timetrack panels and apply personalization we designed for user_edit.php:

- add password2 field
- hide password
- validate
-

on our \user\UserEditPanel.class.php, or directly to generated UserEditFormBase.class.php so the modification is inherited by drafts and our form and panel, but remember that this file will be overwritten by regeneration…

Remember to change also info to template location in
\user\UserEditPanel.class.php

```
// Setup Callback and Template
$this->strTemplate = './UserEditPanel.tpl.php';
```

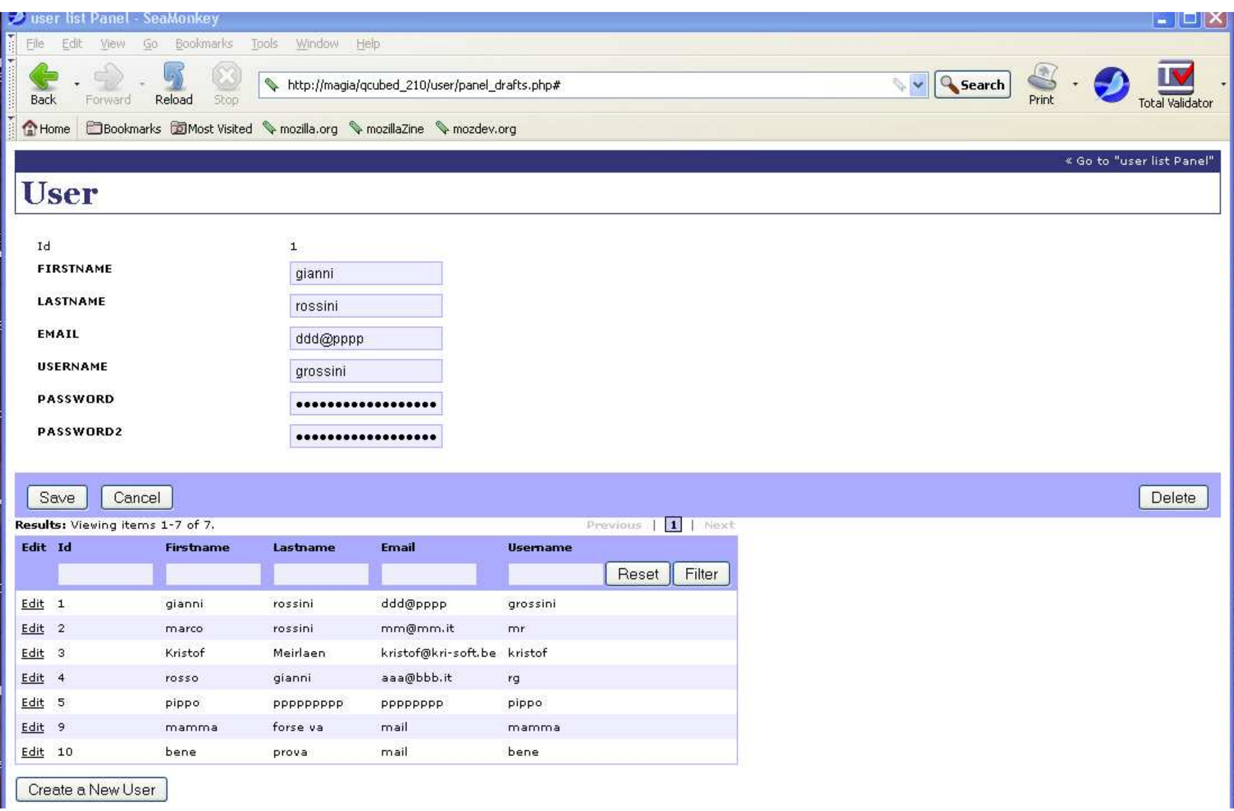
and in \user\UserListPanel.class.php

```
// Setup the Template
$this->Template = './UserListPanel.tpl.php';
```

Obviously save action is performed as well.

I suggest (as exercise) to remove password field from list …

So the panel in action calling http://magia/qcubed_210/user will be (editing gianni)



CHAPTER 07: CREATING THE APPLICATION

We have gone as far as we could with QCubed in terms of code generation. We now have to take what QCubed has created for us, and built upon it.

STEP 1: CREATE THE INITIAL APPLICATION PAGE – INDEX.PHP

We already create our table handler in a complete separate directories under our current QCubed installation. We will use root directory for our application start files.

To do this we rename index.php to index_install.php so we can create our application devoted index php.

To minimize duplication of code, we will split a page into 3 parts

- header.inc.php: header of the page + menu
- index.php: the page itself
- footer.inc.php: included on every page

So every page will look like this:

```
<?include "header.inc.php" ?>
CONTENT
<?  include  "footer.inc.php"  ?>
```

HEADER.INC.PHP

In this sample, I have taken the header from the original index.php from the QCubed framework. I did remove the copyright info, and changed the titles. Feel free to use your own header.

```
<?php require_once('../includes/configuration/prepend.inc.php');?>
 <html>
   <head>
      <title>Time Track demo using Qcubed v.2.0.2</title> <style>
      TD,  BODY  {  font:  12px  <?php  echo  QFontFamily::Verdana;  ?>;  }
      .title  {  font: 30px <?php  echo  QFontFamily::Verdana; ?>; font-weight: bold;
                 margin-left:-2px;}
      .title_action { font: 12px <?php echo QFontFamily::Verdana; ?>; font-weight: bold;
                 margin-bottom: -4px; }
      .item_divider  {  line-height:  16px;  }
      .heading  {  font:  16px  <?php  echo  QFontFamily::Verdana;  ?>;  font-weight:  bold;  }
      </style>
   </head><body>
      <div class="title_action">Time Track demo using Qubed v.2.0.2</div>
      <ul>
      <? include "menu.inc.php" ?>
      </ul>
```

MENU.INC.PHP

Note that the header.inc includes the file “menu.inc.php”. This is where we will create the menu for our application.

```
<a  href="timetrack_edit.php">Add  Time</a>  |  <a  href="changepass.php">Change  password</a>  |
<a href="reports.php">View reports</a> | <a  href="user_list.php">Manage  users</a>|
<a href="project_list.php">Manage projects</a> | <a href="logout.php">Logout</a>
```

FOOTER.INC.PHP

The footer is a simple file, just closing the body and html tags:

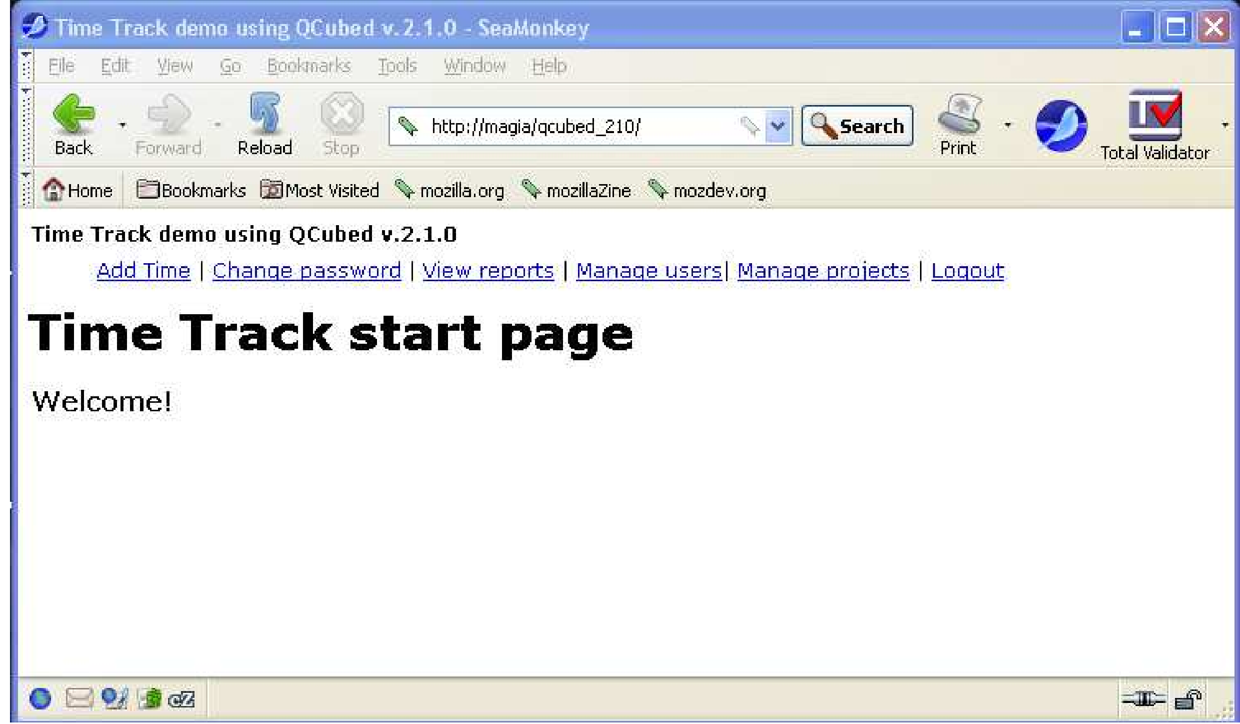
```
   </body><
/html>
```

INDEX.PHP

This is a simple welcome page, and as specified before, we will include the header, the footer and put some content in between:

```
<?  include  "header.inc.php"  ?>

      <div class="title">Time Track start page</div>
      <br class="item_divider" />
      <span  class="heading">Welcome!</span><br  /><br  />

      <br  /><br  /><br  />

<?  include  "footer.inc.php"  ?>
```

Our simple welcome page:



STEP 2: CREATING A LOGIN FORM

QCubed did not generate a login form for us. After all, QCubed a framework and as such doesn't have built-in user authentication. But by using QForms it is extremely easy to build it!

A login form is basically a QForm with 3 controls: a username field, a password field and a login button.

As we know that a QForm is 2 pages, let us create them: login.php for the logic and login.tpl.php for the display.

LOGIN.PHP

```
<?php require_once('./includes/prepend.inc.php');

class LoginForm extends QForm {

   // Local instance of the UserMetaControl
   protected $mctUser;

   // Controls for User's Data Fields
   protected $txtUsername;
   protected $txtPassword;
   protected $btnLogin;

   protected function Form_Create() {

// Use the CreateFromPathInfo shortcut (this can also be done manually using the UserMetaControl constructor)
// MAKE SURE we specify "$this" as the MetaControl's (and thus all subsequent controls') parent
   $this->mctUser = UserMetaControl::CreateFromPathInfo($this);

      // Call MetaControl's methods to create qcontrols based on User's data fields
      $this->txtUsername = $this->mctUser->txtUsername_Create();
      $this->txtPassword = $this->mctUser->txtPassword_Create();

      $this->txtPassword->TextMode  =  QTextMode::Password;

      $this->btnLogin = new QButton($this);
      $this->btnLogin->Text = QApplication::Translate('Login');

      $this->btnLogin->AddAction(new QClickEvent(), new QServerAction('btnLogin_Click'));
      $this->btnLogin->PrimaryButton = true;
   }

      protected  function  btnLogin_Click($strFormId,  $strControlId,  $strParameter)
   {
        }
}
LoginForm::Run('LoginForm',  'login.tpl.php');
?>
```

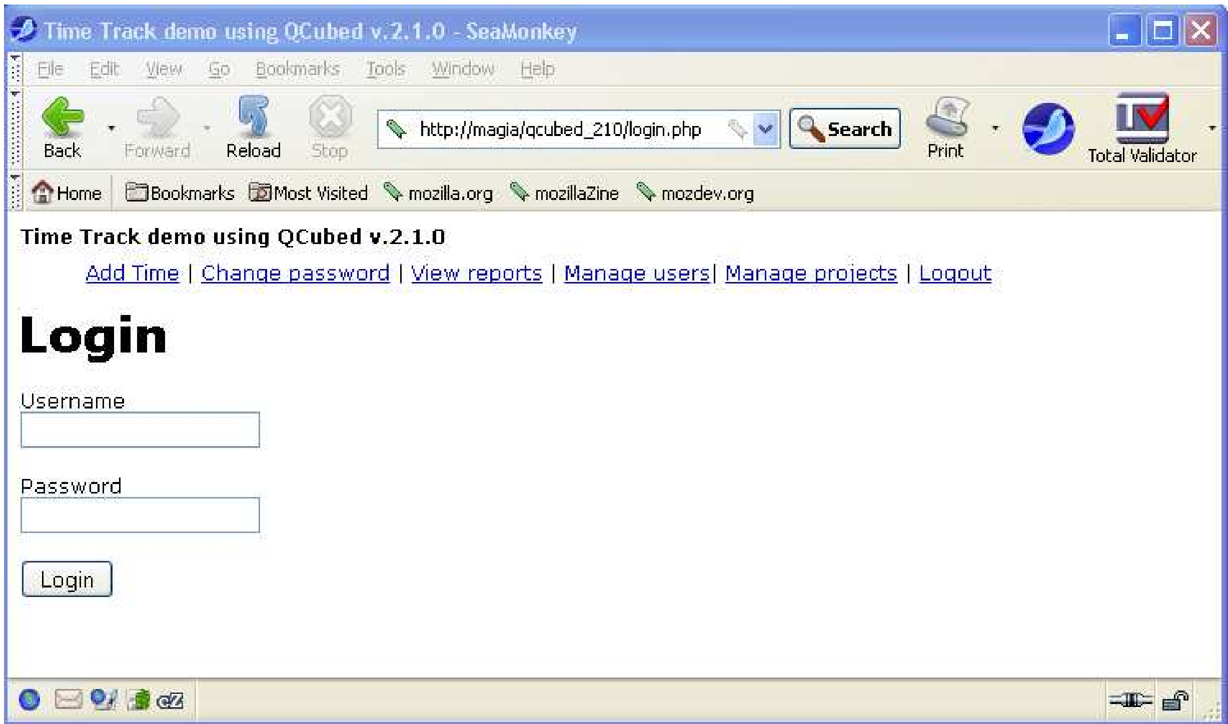
Nothing new here: we created a class LoginForm, which extends from QForm. We assign 3 members to it to hold the fields and button, and created default values for it. We also assigned a “btnLogin_Click” event to the btnLogin. At the end, we “run” the form.

LOGIN.TPL.PHP

```
<?include "header.inc.php" ?>
   <?php $this->RenderBegin() ?>
      <div class="title">Login</div>
      <br class="item_divider" />
      <?php $this->txtUsername->RenderWithName(); ?>
      <br class="item_divider" />
      <?php $this->txtPassword->RenderWithName(); ?>
      <br class="item_divider" />
      <?php  $this->btnLogin->Render()  ?>
   <?php $this->RenderEnd() ?>
<? include "footer.inc.php" ?>
```

Nothing special on this page either: we include the header and footer, and for the QCubed framework to do it's job, include RenderBegin() and RenderEnd(). Then, for each member we declared in the login.php page, we call “RenderWithName”. The difference with Render() is that now the name that we have assigned to the object in the form_create will be added as a label.

Going to page http://magia/qcubed_210/login.php with browser, the form is created but, as expected, do nothing:



STEP 3: VERIFYING THE USER

The form we have created so far does not do anything, as the btnLogin_Click() method is still empty.

To check if a valid user has authenticated, we need to retrieve the user from the database, and verify the password against the (hashed) value that is present in the database.

Let's put this into code:
Login.php (insert code on btnLogin_Click)

```
protected  function  btnLogin_Click($strFormId,  $strControlId,  $strParameter)
{
// chapt. 7  step 2 verifying the user
  $objUser = User::LoadByUsername($this->txtUsername->Text);
  if (!$objUser || $objUser->Password != sha1($this->txtPassword->Text))
  {
     $this->txtPassword->Text = "";
     $this->txtPassword->Warning = "Unknown user or password";
     return;
  }

  $_SESSION['User'] = serialize($objUser);
  QApplication::Redirect('index.php');
}
```

First, we load the user by it's username. As the username had an UNIQUE index on it, the LoadByUsername function was generated by the QCubed codegen framework.

Next, we check if the user object is filled. If not, no valid username was specified. We set a warning for the user, and return.

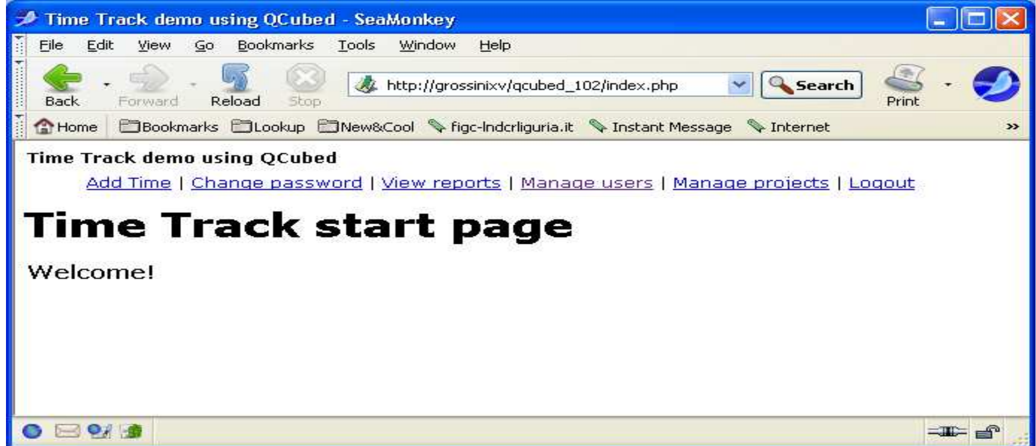
Next, we verify if the password we received matches the one in the database. If not, simply set a warning and return.

If all checks pass, we know we have a valid user, and we can redirect him to the index.php page.

We also store the user object in the plain PHP session variable. This allows us to verify if a user is logged on or not later.

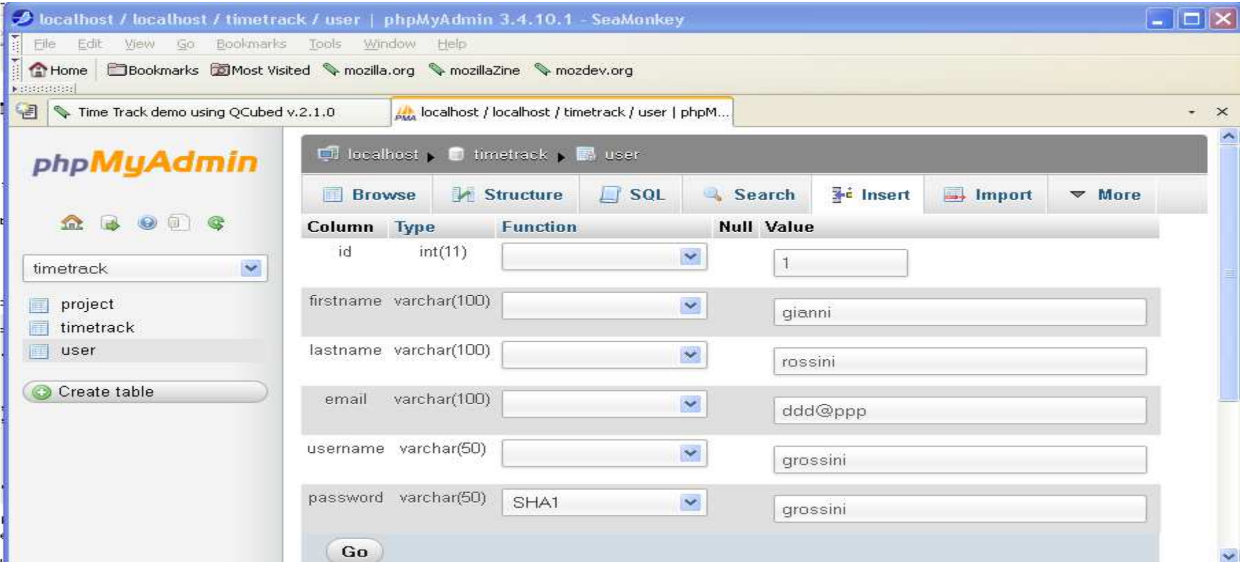
Going now to page http://magia/qcubed_210/login.php with browser (now button does his work !!!!):

This is result of login with correct user and password:



And if user or password were worng:

Survival –password trick: *The user and password im my test of gianni rossini entry where grossini, if you have some problem with data present in user table for this entry you can use* ***phpmyadmin*** *to edit the record and update password field using sha1 function*



STEP 4: CREATING A PROTECTED PAGE

There are many ways to protect pages from unauthorized access. One way to do it is to verify on every page that needs to be protected whether a user is logged in or not.

To avoid duplication of code, we create a special file which handles this:

PROTECTED.INC.PHP

```
<?
   require_once("includes/configuration/prepend.inc.php");
   if (!isset($_SESSION['User']))
      QApplication::Redirect('login.php');

   $objUser  =  unserialize($_SESSION['User']);
// make sure no errors occured in translation and the session's User variable is a user object

   if (!($objUser instanceof User))
      QApplication::Redirect('login.php');
?>
```

What this file does it to check if the current php session contains the variable “User”. If not, it redirects to the login.php page.

We then take the value of the $_SESSION['User'] and put it in the $objUser. If the result works out ok, we have a valid user. If not, we redirect to the login page.

Now, on every page we wish to password protect, simply include “protected.inc” and the user will be redirected to login.php when needed.

So, our index.php now becomes (as we did before without knowing why):

```
<? include "protected.inc.php" ?>
<? include "header.inc" ?>
...
```

And we should be able to login as well!

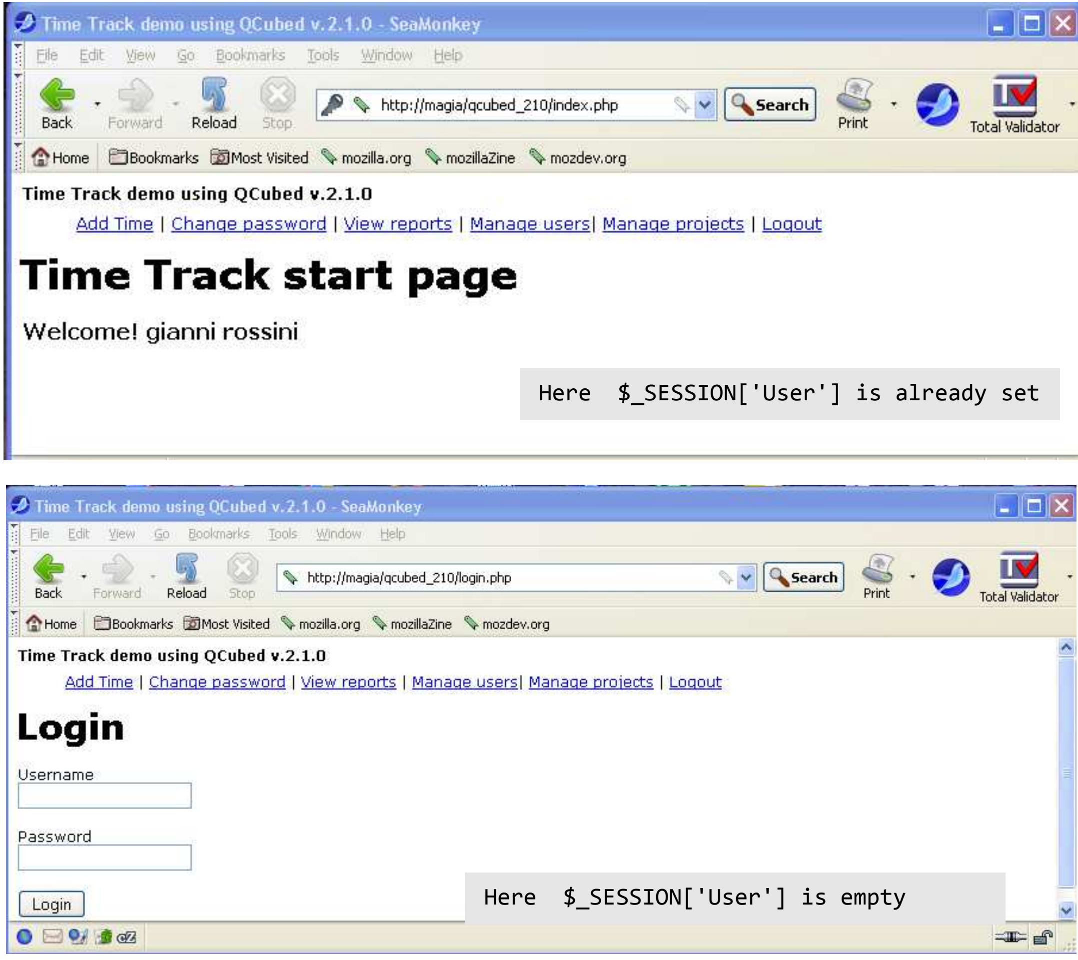
NOTE: Pay attention that in our test path we already did a login before protected page step 3; so our `$_SESSION['User']` can be already set.

To realize this we added in index.php, after **welcome!** the content of `$_SESSION['User'].`

```
<span  class="heading">Welcome!
<? if (isset($_SESSION['User'])) {$temp = unserialize($_SESSION['User']); echo $temp ;}?>
</span><br  /><br  />
```

Unset of **($_SESSION['User']** is done in logout.php (next we see it).

You can also get session unsetted stopping and restarting Apache or when session expiration time is elapsed.



STEP 5: LOGOUT

Logout can be very simple: just unset the $_SESSION["User"] variable and redirect to the login page:

```
<?php require_once('/includes/configuration/prepend.inc.php');
class  LogoutForm  extends  QForm  {
   protected $btnLogout;
   protected function Form_Create() {
      $this->btnLogout = new QButton($this);
      $this->btnLogout->Text = QApplication::Translate('confirm logged_out');
      $this->btnLogout->AddAction(new QClickEvent(), new QServerAction('btnLogout_Click'));
      $this->btnLogout->PrimaryButton = true;
   }
   protected function btnLogout_Click($strFormId, $strControlId, $strParameter) {
      if (isset($_SESSION['User']))
      unset($_SESSION['User']);
      QApplication::Redirect('index.php');
   }
}
LogoutForm::Run('LogoutForm',  'logout.php.inc');
?>
```

logout.tpl.php

```
<?include "header.inc.php" ?>
   <?php $this->RenderBegin() ?>
      <?php $this->btnLogout->Render() ?>
      <?php $this->RenderEnd() ?>
<? include "footer.inc.php" ?>
```

STEP 6 :ADD TIME PAGE

Now that we have created our authentication pages, we can start working on the remaining pages.

The first page in our menu is the "Add time" page. This is basically the "timetrack_edit.php" page that was generated from QCubed during the code generation. So... let us take this page out of the code generation and put it into our "timetrack" directory as instructed in chapter 05, apply also the mod we learned. Remember, our QCubed form has 2 pages, so we need to take the "timetrack_edit.tpl.php" as well. Remember to also "tell" timetrack_edit.php the new location of timetrack_edit.tpl.php in the Run method.

This work is already done and in place, if you arrive here after Chapter 05 study session.

For now, we have the following files:

```
Directory di C:\wamp\www\qcubed_210\timetrack
02/10/2010  17.24            1.686 form_drafts.php
04/10/2010  21.36            3.454 timetrack_edit.php
02/10/2010  10.48            1.270 timetrack_edit.tpl.php
04/10/2010  12.29            2.403 timetrack_list.php
02/10/2010  18.21              954 timetrack_list.tpl.php
```

GO MAGICALLY TO TIMETRACK PANEL(EXCERPT FROM CHAPTER 05)

In previous chapter we interacted with Timetrack table accessing to `index_form.php` located in `\timetrack`. We did also some modification to `timetrack_edit.php`, `timetrack_list.php`, `timetrack_edit.tpl.php` and `timetrack_list.tpl.php` to remain in `\timetrack` using a general purpose `form_drafts.php`.

We suggested also that we can work with Panels using an `index.php` who redirect to `panel_drafts.php` and `panel.drafts.tpl.php` derived from those present in from \drafts\panels with some magic added code, so our coding activity will be easy and no modification (or little and easy) were required to generated table related Panels.

Now is time to reuse the magic couple.

INDEX.PHP (MAGIC GENERIC CODE LOCATED IN TABLE RELATED DIRECTORY)

```
<?php
header("Location: ./panel_drafts.php");
?>
```

panel.drafts.php *(generic code located in table related directory as explained in chapt 5)*

```
<?php
   require_once('../qcubed.inc.php');

   // Security check for ALLOW_REMOTE_ADMIN
   // To allow access REGARDLESS of ALLOW_REMOTE_ADMIN, simply remove the line below
   QApplication::CheckRemoteAdmin();

   // Let's "magically" determine the list of genereated Class Panel Drafts by
   // just traversing through this directory, looking for "*ListPanel.class.php" and "*EditPanel.class.php"

   // Obviously, if you are wanting to make your own dashbaord, you should change this and use more
   // hard-coded means to determine which classes' paneldrafts you want to include/use in your dashboard.
   $objDirectory = opendir(dirname(__FILE__));

   $strClassNameArray = array();
   while ($strFile = readdir($objDirectory)) {
      if ($intPosition = strpos($strFile, 'ListPanel.class.php')) {
         $strClassName = substr($strFile, 0, $intPosition);
         $strClassNameArray[$strClassName] = $strClassName . 'ListPanel';
         require( $strClassName . 'ListPanel.class.php');
         require( $strClassName . 'EditPanel.class.php');
      }
   }

   class Dashboard extends QForm {
      protected $lstClassNames;

      protected $lblTitle;
      protected $pnlList;
      protected $pnlEdit;

      protected function Form_Create() {
         $this->lblTitle = new QLabel($this);
         $this->lblTitle->Text = 'AJAX Dashboard';

         $this->pnlList = new QPanel($this, 'pnlList');
         $this->pnlList->AutoRenderChildren = true;

         $this->pnlEdit = new QPanel($this, 'pnlEdit');
         $this->pnlEdit->AutoRenderChildren = true;
         $this->pnlEdit->Visible = false;

/* define list panel */

         $this->pnlList->RemoveChildControls(true);
         $this->pnlEdit->RemoveChildControls(true);
         $this->pnlEdit->Visible = false;

/* call to output list panel */
         global $strClassNameArray;
         global $strClassName;

         //  $this->lstClassNames_Change();
         $this->pnlList->RemoveChildControls(true);
         $this->pnlEdit->RemoveChildControls(true);
         $this->pnlEdit->Visible = false;

         $this->lblTitle->Text = $strClassName;
         $objNewPanel = new $strClassNameArray[$strClassName]($this->pnlList, 'SetEditPane', 'CloseEditPane');
   }
```

```
    /**
     * This Form_Validate event handler allows you to specify any custom Form Validation rules.
     * It will also Blink() on all invalid controls, as well as Focus() on the top-most invalid control.
     */
    protected function Form_Validate() {
       // By default, we report that Custom Validations passed
       $blnToReturn = true;

       // Custom Validation Rules
       // TODO: Be sure to set $blnToReturn to false if any custom validation fails!

       $blnFocused = false;
       foreach ($this->GetErrorControls() as $objControl) {
          // Set Focus to the top-most invalid control
          if (!$blnFocused) {
             $objControl->Focus();
             $blnFocused = true;
          }
          // Blink on ALL invalid controls
          $objControl->Blink();
       }
       return $blnToReturn;
    }
    public function CloseEditPane($blnUpdatesMade) {
       // Close the Edit Pane
       $this->pnlEdit->RemoveChildControls(true);
       $this->pnlEdit->Visible = false;

       // If updates were made, let's "brute force" the updates to the screen
       // by just refreshing the list pane altogether
       if ($blnUpdatesMade)
          $this->pnlList->Refresh();
    }

    public function SetEditPane(QPanel $objPanel = null) {
       $this->pnlEdit->RemoveChildControls(true);
       if ($objPanel) {
          $objPanel->SetParentControl($this->pnlEdit);
          $this->pnlEdit->Visible = true;
       } else {
          $this->pnlEdit->Visible = false;
       }
    }
 }
 Dashboard::Run('Dashboard');
?>
```

panel.drafts.tpl.php *(generic code located in table related directory as explained in chapt 5 whit some mod)*

```
<?php require (__CONFIGURATION__.  '/header.inc.php'); ?>
<?php $this->RenderBegin(); ?>
   <div id="titleBar">
      <h2 id="right"><a href="../index.php">&laquo; <?php _t('Go to "Menu"'); ?></a></h2>
      <h2>Panel from Qcodo - by ma.gi.a. di Rossini</h2>
      <h1><?php $this->lblTitle->Render(); ?></h1>
   </div>
   <div id="right">
      <?php $this->pnlEdit->Render(); ?>
      <?php $this->pnlList->Render(); ?>
   </div>
   <br clear="all" style="clear:both" />
<?php $this->RenderEnd(); ?>
<?php require(__CONFIGURATION__ . '/footer.inc.php'); ?>
```

Now we can edit menu.inc.php to modify link to our timetrack related directory

menu.inc.php

From :

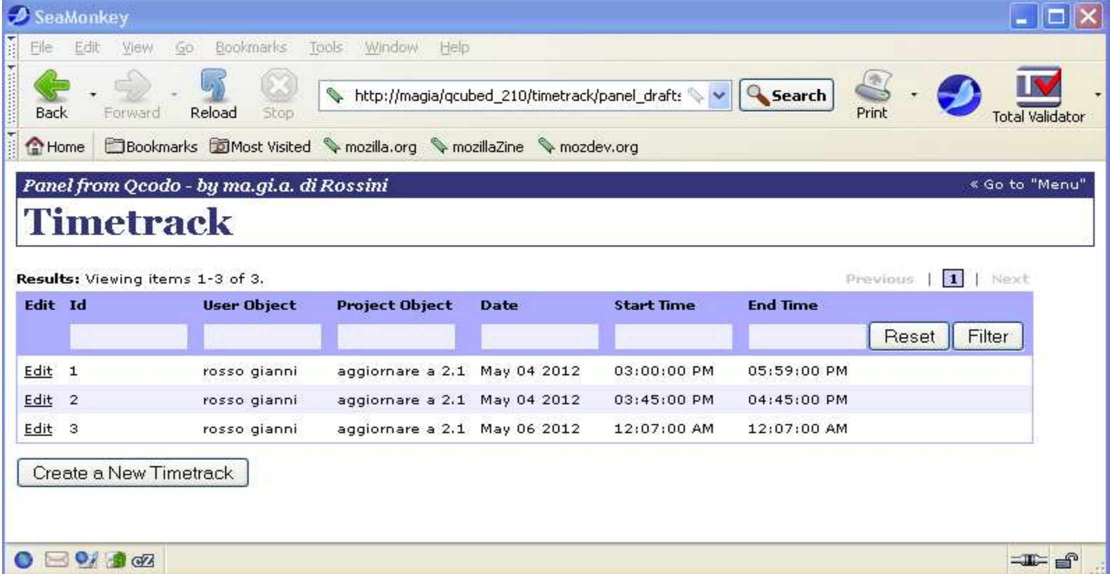
```
<a href="timetrack_edit.php">Add Time</a> |
```

To :

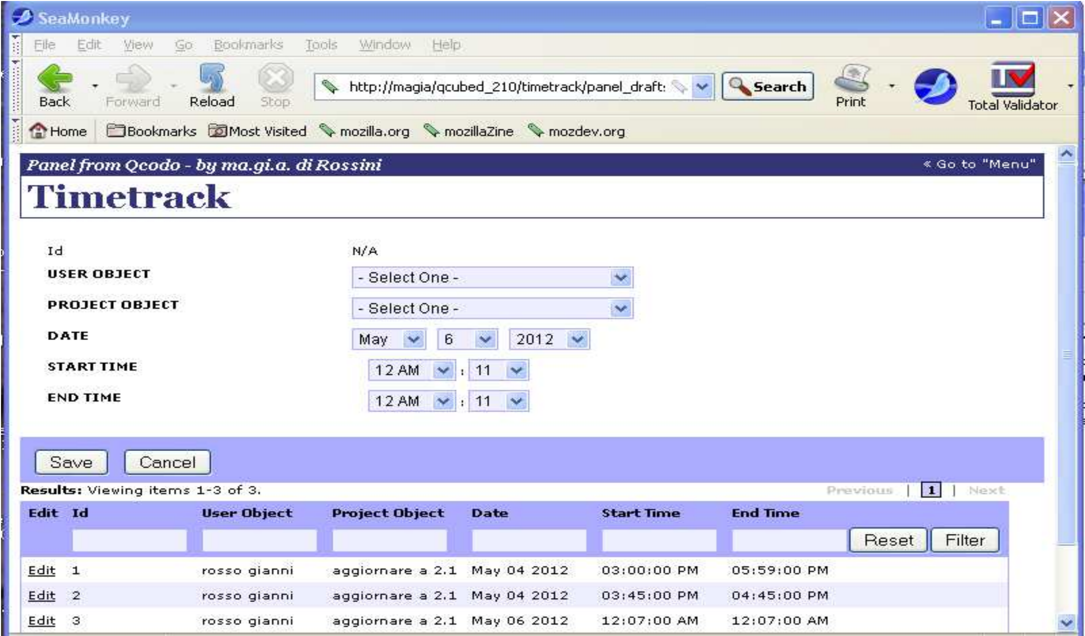
```
<a href="timetrack/index.php">Add Time</a> |
```

You will see here that all modification we did on metacontrol related timetrack (setting as default actual date and time) are magically retained and will be retained also after a code regeneration.

You will see here that all modification we did on metacontrol related timetrack (setting actual date and time as default) are magically retained and will be retained also after a code regeneration. The result page will be:



And pushing new timetrack creation:



STEP 7: MODIFYING TIMETRACK PANEL EDIT

The timetrack page contains some items that should not be present. This is the case with "ID" and with the "User Object". For the user object, we don't need a drop down list: we know the current user, and this value should be used.

We also want to protect the page. This can be done by copying the logic we included in "protected.inc.php" in our magic `panel_drafts.php` . Let's start with this:

TIMETRACK/PANEL_DRAFTS.PHP

```
<?php
   // Include prepend.inc to load Qcubed
   // magia 2010
   require('../includes/configuration/prepend.inc.php');
   //  go to login page if no user logged on
   if (!isset($_SESSION['User']))
      QApplication::Redirect('../login.php');

   $objUser = unserialize($_SESSION['User']);

   // make sure no errors occured in translation and the session's User variable is a user object
   if (!($objUser instanceof User))
      QApplication::Redirect('../login.php');
...
```

Let us also remove the unwanted items from TimetrackEditPanel.tpl.php: remove the following lines:

TIMETRACKEDITPANEL.TPL.PHP

```
<?php //$this->lblId->RenderWithName(); ?>

<?php //$this->lstUserObject->RenderWithName(); ?>
```

Back to TimetrackEditPanel.php

In the previous chapter, we added the function Form_Validate to perform some validation. Note that this validation is present in timetrack_edit.php copied from drafts (the Form) but not present in Panel so we need to add this function to our Panel code in Function Validate() (this is already done if we come here from previous chapter).

```
      // by magia chapter 07 step 7 (se also chapter 5 for validation rule)
      public function Validate() {
         $blnToReturn = true;
         if ($this->calStartTime->DateTime > $this->calEndTime->DateTime){
            $this->calEndTime->Warning = "Start time must be smaller then endtime";
            $blnToReturn =  false;
         }
         return $blnToReturn;
      }
```

Also remember that in the previous chapter, we worked on metacontrols to insert a function named `SaveUser ()` to update the password to contain the hash instead of the plain text.

In the same way for Timetrack class, we can insert in metacontrols a function called “SaveTimetrack”. This time we will modify this function so that it does not take the user from the list box (we have removed it in the template), but to take the value from the session.

We can do this by overriding the `SaveTimetrack()` function to set the correct value for the User:

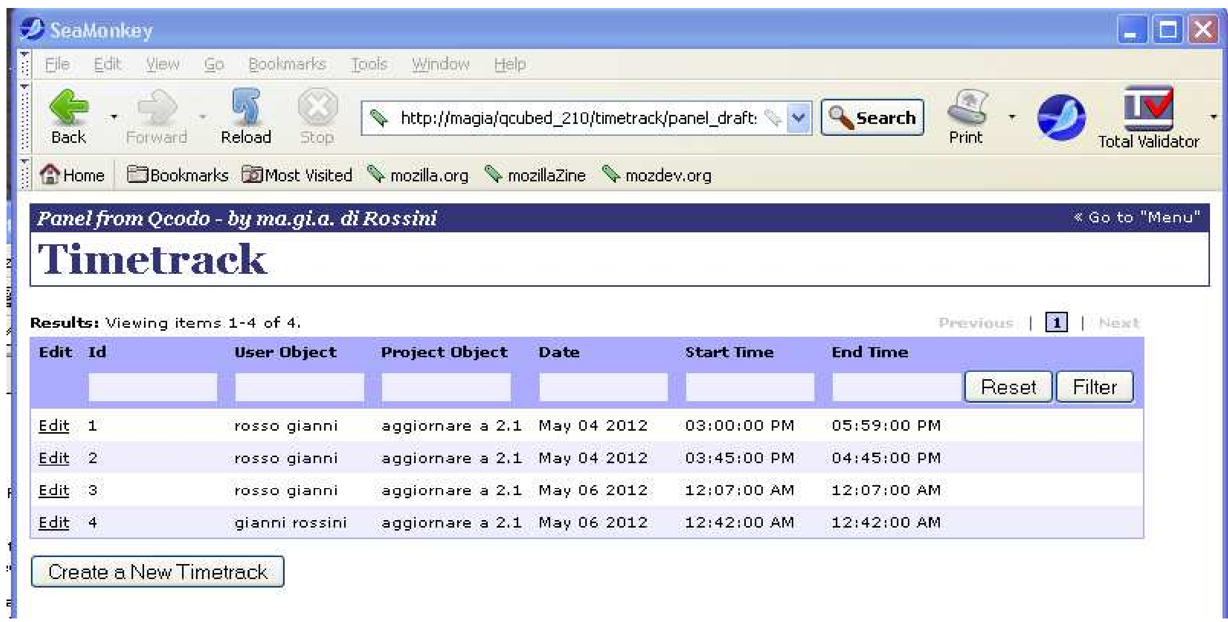
TIMETRACKMETACONTROL.CLASS.PHP (**IN INCLUDES/META_CONTROLS**)

```
// added by magia 2012 chapter 07 step 7
   public function SaveTimetrack() {
         $objUser = unserialize($_SESSION['User']);
         $this->objTimetrack->User = $objUser->Id;
         // we must unset lstUserObject so in parent function the relative line
         //   if   ($this->lstUserObject)   $this->objTimetrack->User   =   $this->lstUserObject->SelectedValue;
         // has no effect on $this->objTimetrack->User
         $this->lstUserObject = False;
         parent::SaveTimetrack();
   }
}
```

We could copy here all the function `SaveTimetrack()` and comment out the line related to setting User from list box, but I proposed a more elegant way to override the parent function. *But there is a Side effect*: if you do not unset `$this->lstUserObject`, you will have an error due the fact that user cannot be null in table.

Also notice that in TimetrackEditFormBase.class.php, after the data is saved to the database, we hide Edit Panel and show only the updated list Panel.

Adding time tracking functionality is now complete (in entry 4 user is from login – no edit field present in edit Panel).



Survival hint –If in your editing activity your editor create some .bak file with the same name, the index.php will be confused by two class with the same name and will print this message:

```
Fatal error: Cannot redeclare class TimetrackListPanel in
C:\wamp\www\qcubed_102\timetrack\TimetrackListPanel.class.php on line 106
```

No problem: simple delete the bak file or rename it and retry.

Step 8: Change password page

This is yet another page that can be based of a QForm. So you know what to do: create 2 files. In changepass.php: put in the fields you need, and assign an event to the button, and in changepass.tpl.php render the fields on the position you want them.

CHANGEPASS.PHP

```
<?php
   include('protected.inc.php');

   class ChangepassForm extends QForm {
      protected $txtOldPassword;
      protected $txtPassword;
      protected $txtPassword2;
      protected $btnSave;

      protected function Form_Create() {
         $this->txtOldPassword = new QTextBox($this);
           $this->txtOldPassword->Name = 'oldpassword';
           $this->txtOldPassword->TextMode  =  QTextMode::Password;
         $this->txtPassword = new QTextBox($this);
           $this->txtPassword->Name = 'password';
           $this->txtPassword->TextMode = QTextMode::Password;
         $this->txtPassword2 = new QTextBox($this);
           $this->txtPassword2->Name = 'password2';
           $this->txtPassword2->TextMode = QTextMode::Password;

         $this->btnSave = new QButton($this);
         $this->btnSave->Text = QApplication::Translate('Change');
         $this->btnSave->AddAction(new QClickEvent(), new QServerAction('btnSave_Click'));
         $this->btnSave->PrimaryButton = true;

      protected  function  btnSave_Click($strFormId,  $strControlId,  $strParameter)  {
      }
   }
   ChangepassForm::Run('ChangepassForm',  'changepass.tpl.php');
?>
```

Nothing new, again... : we defined 3 input fields (now they are all 'password style' boxes), and one button. We also assigned an event to the button, made it a primary button and have set the “CausesValication” to true in order to trigger the Form_Validate function when the button is clicked.

CHANGEPASS.TPL.PHP

```
<?include "header.inc.php" ?>
   <?php $this->RenderBegin() ?>

   <div class="title">Change password</div> <br class="item_divider" />

      <?php $this->txtOldPassword->RenderWithName(); ?> <br class="item_divider" />
      <?php $this->txtPassword->RenderWithName(); ?> <br class="item_divider" />

      <?php $this->txtPassword2->RenderWithName(); ?> <br class="item_divider" />
      <?php  $this->btnSave->Render()  ?>

   <?php $this->RenderEnd() ?>
<? include "footer.inc.php" ?>
```

We will now add validation using Form_Validate: first, we check if the old password matches the new password. If it doesn't we inform the user. Next, we need to verify if the 2 passwords that were entered are the same. If not, we inform the user. If all succeeds, we return true, and we store the hashed value of the password in the database by calling Save() in the btnSave_Click() function. Last but not least, we update the session information and redirect the user to the start page.

CHANGEPASS.PHP

```
  $this->btnSave->CausesValidation = true;


        }
        protected function Form_Validate() {
            $objUser = unserialize($_SESSION['User']);
            if ($objUser->Password != sha1($this->txtOldPassword->Text)) {
// magia 2009
            //  if ($objUser->Password != $this->txtOldPassword->Text) {
               $this->Clear_PasswordBoxes();
               $this->txtOldPassword->Warning = "Incorrect old password";
               return;
            }
            if ($this->txtPassword->Text != $this->txtPassword2->Text) {
               $this->Clear_PasswordBoxes();
               $this->txtPassword2->Warning = "Passwords do not match";
               $this->txtPassword->Warning = "Passwords do not match";
               return;
            }
            return true;
        }
        protected function Clear_PasswordBoxes() {
            $this->txtPassword->Text = "";
            $this->txtPassword2->Text = "";
            $this->txtOldPassword->Text = "";
        }
        protected function btnSave_Click($strFormId, $strControlId, $strParameter) {
            $objUser = unserialize($_SESSION['User']);
            $objUser->Password = sha1($this->txtPassword->Text);
// magia 2009
            //$objUser->Password = $this->txtPassword->Text;
            $objUser->Save();
            $_SESSION['User'] = serialize($objUser);
            QApplication::Redirect('index.php');
        }...
```

STEP 9: MANAGE USERS AND PROJECTS

In case the user is logged in as “admin”, that user can manage users and projects. This is simple, because... all the functionalities to do this have been code-generated by QCubed! We just make sure that our menu links to them. We could simply link to the drafts or panels. But we can also move the panels files related to project and user to our created \project and \user directories, adding also the magic index.php, panel_drafts.php and panel_drafts.tpl.php.

To edit projects, we need to have the following files (from the drafts\panels folder):

```
Directory di C:\wamp\www\qcubed_210\project
06/10/2010  20.14                49 index.php
06/10/2010  20.14             4.990 panel_drafts.php
06/10/2010  15.35               544 panel_drafts.tpl.php
04/10/2010  19.05             4.035 ProjectEditPanel.class.php
04/10/2010  19.05               738 ProjectEditPanel.tpl.php
04/10/2010  19.05             3.713 ProjectListPanel.class.php
04/10/2010  19.05               414 ProjectListPanel.tpl.php

Directory di C:\wamp\www\qcubed_210\user
02/10/2010  17.24             1.734 form_drafts.php
06/10/2010  20.14                49 index.php
06/10/2010  20.14             4.990 panel_drafts.php
06/10/2010  15.35               544 panel_drafts.tpl.php
04/10/2010  19.05             5.306 UserEditPanel.class.php
04/10/2010  19.05               951 UserEditPanel.tpl.php
04/10/2010  19.05             3.736 UserListPanel.class.php
04/10/2010  19.05               408 UserListPanel.tpl.php
```

(note that in UserEditPanel we will apply changes we did in user_edit form)

Apply some change to link in menu.inc.php :

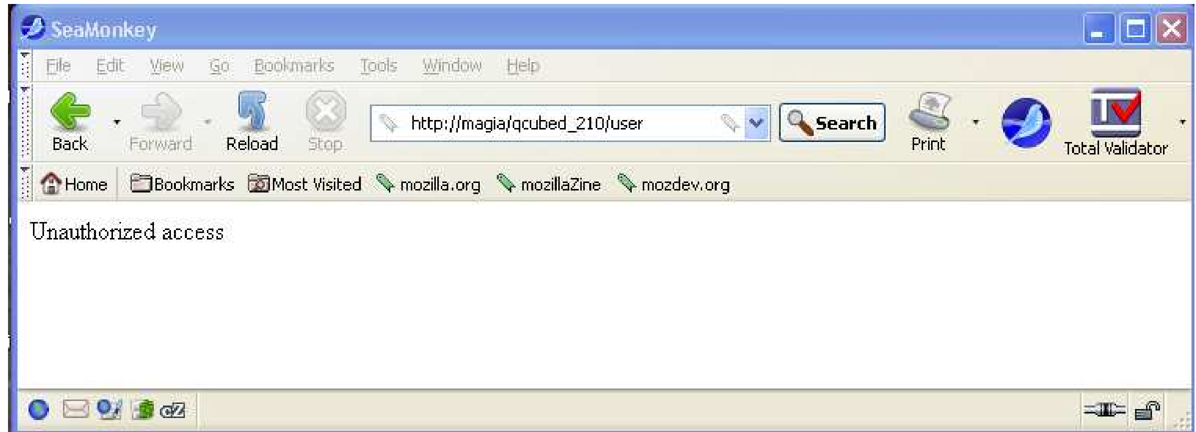
```
<a href="timetrack/index.php">Add Time</a> |
<a href="changepass.php">Change password</a> |
<a href="reports.php">View reports</a> |
<a  href="user/index.php">Manage users</a>  |
<a href="project/index.php">Manage projects</a> |
<a href="logout.php">Logout</a>
```

And we can go from menu to add/modify/delete the users and projects.
Finally, let us also protect all these pages so that they can only be accessed by the “grossini” user(who can administer, insert or modifiy user and project)
We do this by simply, in addition to the regular login check, also verify the user name.

```
//  go to login page if no user logged on
if (!isset($_SESSION['User']))
QApplication::Redirect('../login.php');

$objUser = unserialize($_SESSION['User']);
   // make sure no errors occured in translation and the session's User variable is a user object
   if (!($objUser instanceof User))
   QApplication::Redirect('../login.php');
   //

if ($objUser->Username != "grossini") {
   echo "Unauthorized access";
   exit;
}
```

We then include this test in panel_drafts.php located in \project and \user.

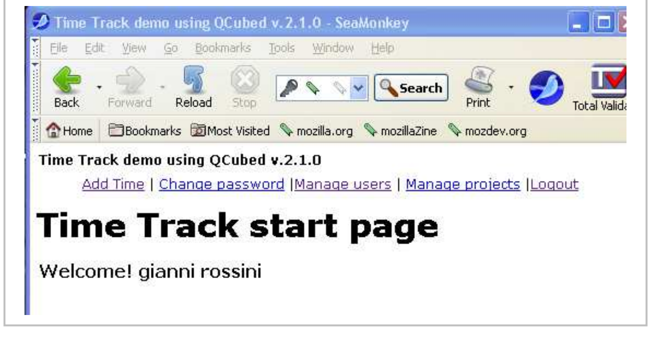
After you have done this, and you are not logged in as administrator, if you tray ta access directly to http://magia/qcubed_210/user you will see the following:



Note that the menu is still available for everybody. If you want that cleaned up, just include some php code into the menu.inc, for example:

MENU.INC

```
<?php
     if (isset($_SESSION['User'])){
      $objUser  =  unserialize  ($_SESSION['User']);
      if  (($objUser->Username)){
         echo '<a href="timetrack/index.php">Add Time</a> |
            <a href="changepass.php">Change password</a> |';
               // this user is able to admin user and project
               // same control in \user\panel_drafts.php and \project\panel_drafts.php
         if  (($objUser->Username)== 'grossini'){
            echo '<a  href="user/index.php">Manage users</a>  |
                   <a  href="project/index.php">Manage  projects</a>  |';
         }
      echo '<a href="logout.php">Logout</a>';
      }
  } ?>
```



gianni rossini can admin also user and project

marco rossini can enter timetrack and change his password

You can also adjust the link in generic `panel_drafts.tpl.php` to return to initial menu:

```
<?php $strPageTitle="Panel Drafts" ?>
<?php require(__CONFIGURATION__ . '/header.inc.php'); ?>
<?php $this->RenderBegin(); ?>
   <div id="titleBar">
     <h2 id="right"><a href="../index.php">&laquo; Go to "menu"</a> </h2>
     <h2> </h2>
     <h1><?php $this->lblTitle->Render(); ?> </h1>
     </div>
   <div id="right">
         <?php $this->pnlEdit->Render(); ?>
         <?php $this->pnlList->Render(); ?>
   </div>
   <br clear="all" style="clear:both" />
<?php $this->RenderEnd(); ?>
<?php require(__CONFIGURATION__ . '/footer.inc.php'); ?>
```

Summary

Using this simple guidelines spiegate in this chapter you will be able to expand your application with less effort:

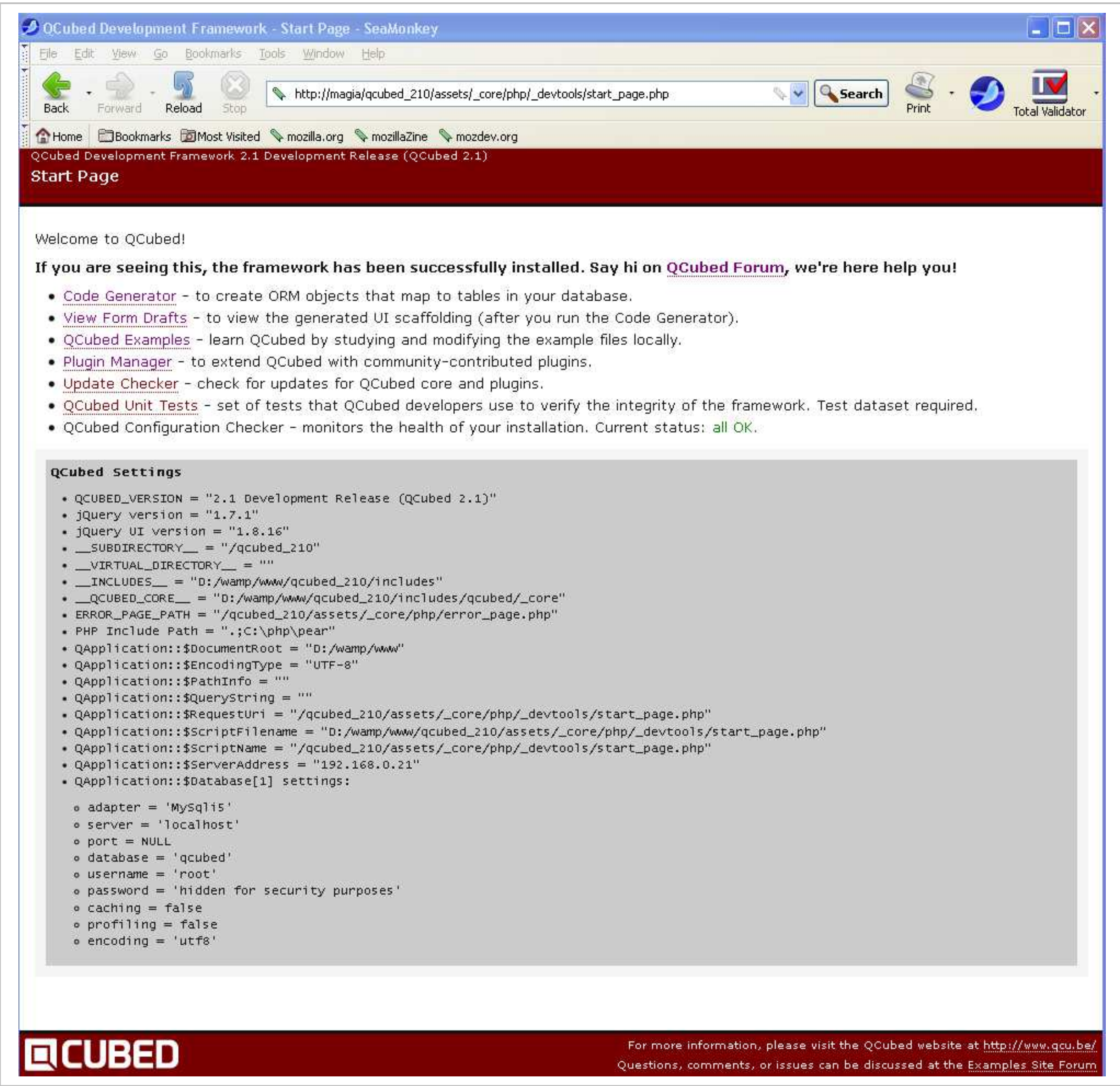
- Create a dir for every table
- Copy there from drafts\panels the table_related Panels (list and edit)
- Copy there the magic index.php, panel_drafts.php and panel_drafts.tpl.php
- Link to dir table_related from menu

And … let your db grow with…your live data…..

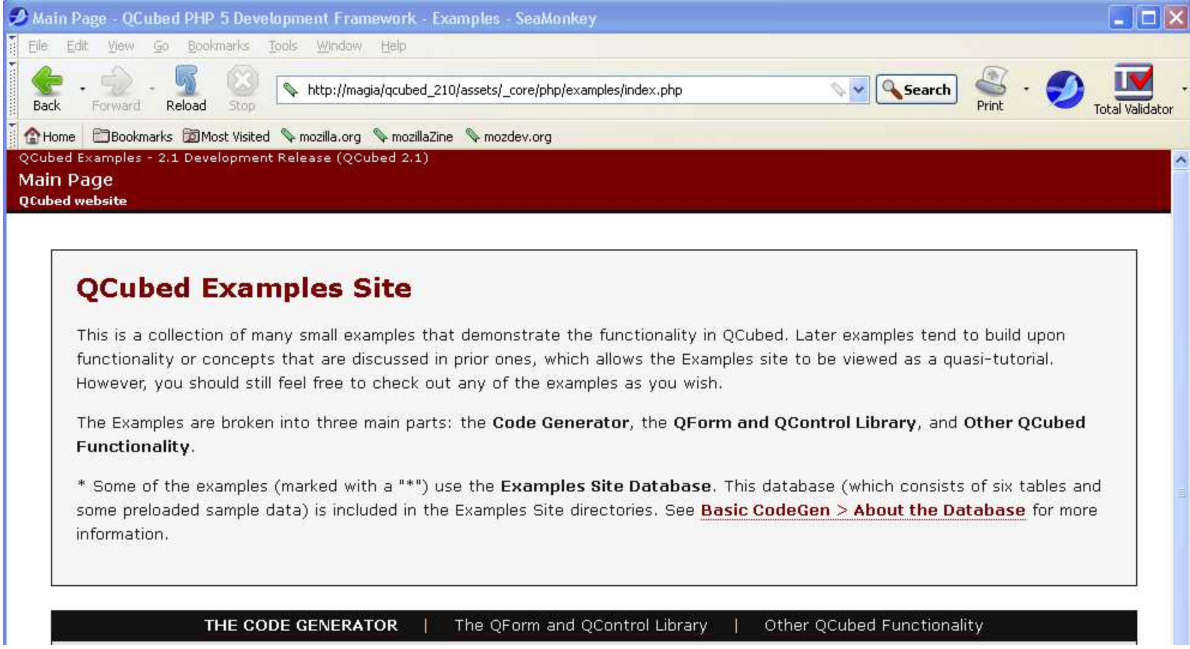
CHAPTER 8: QCUBED ACTIONS AND EVENTS

Actions and events play a very important role in QCubed. This chapter will get into the details of the event handling within QCubed.

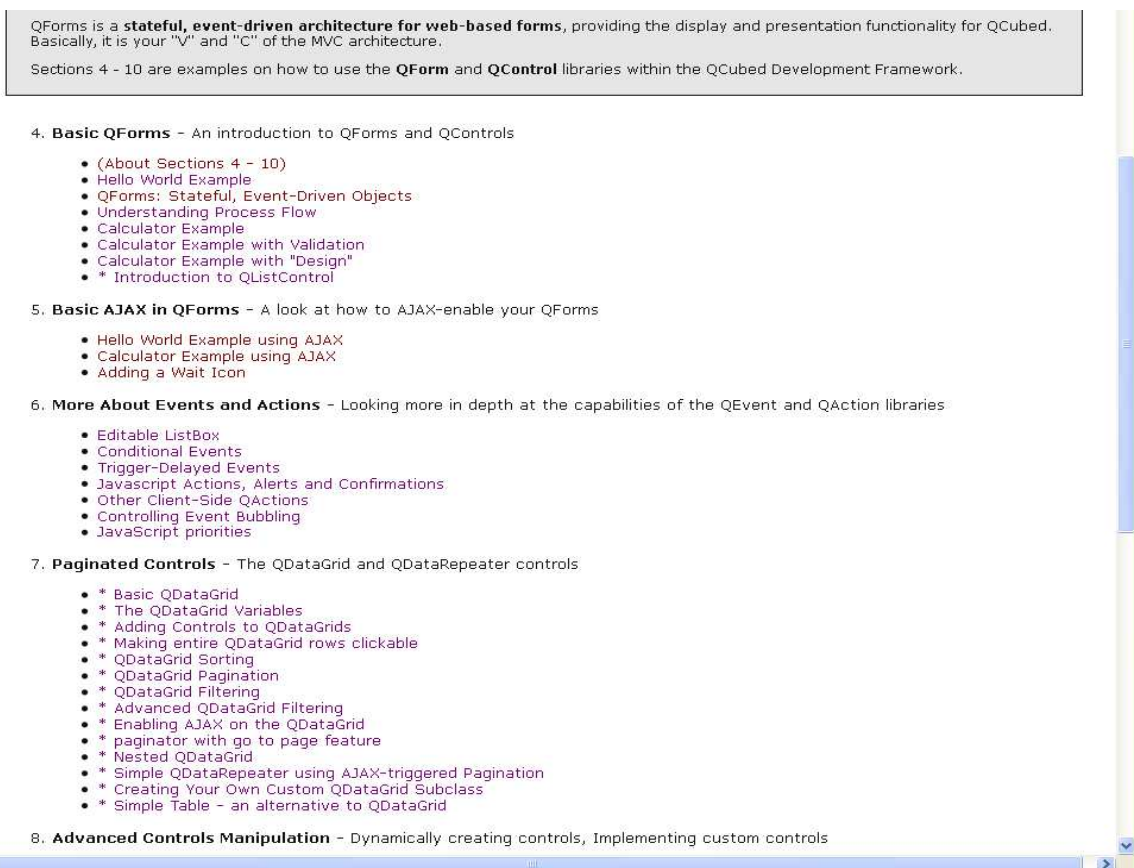
In this chapter we will follow the quidance of quasi-tutorial Examples we downloaded with our Qcodo. Examples are linked from initial local installation welcome page we renamed in index_install.php.



The Examples are broken into three main parts: the **Code Generator**, the **QForm and QControl Library**, and **Other QCubed Functionality**.



To go in depth with Qcubed Actions and Events we will use **QForm and QControl Library** part section 4 – 5 – 6.



BASIC QFORM - A LOOK OVER QFORMS AND QCONTROLS AGAIN

Remember our first QForms application in chapter 2. Using Qcubed class we was able to create a simple ajax interactive application with very low line of code.

What happen behind the scene?

UNDERSTANDING STATE

When you clicked on the button, the form actually posted back to itself. However, the state of the form was remembered from one webpage view to the next.

This is known as **FormState.**
QForm objects, in fact, are stateful objects that maintain its state from one post to the next.

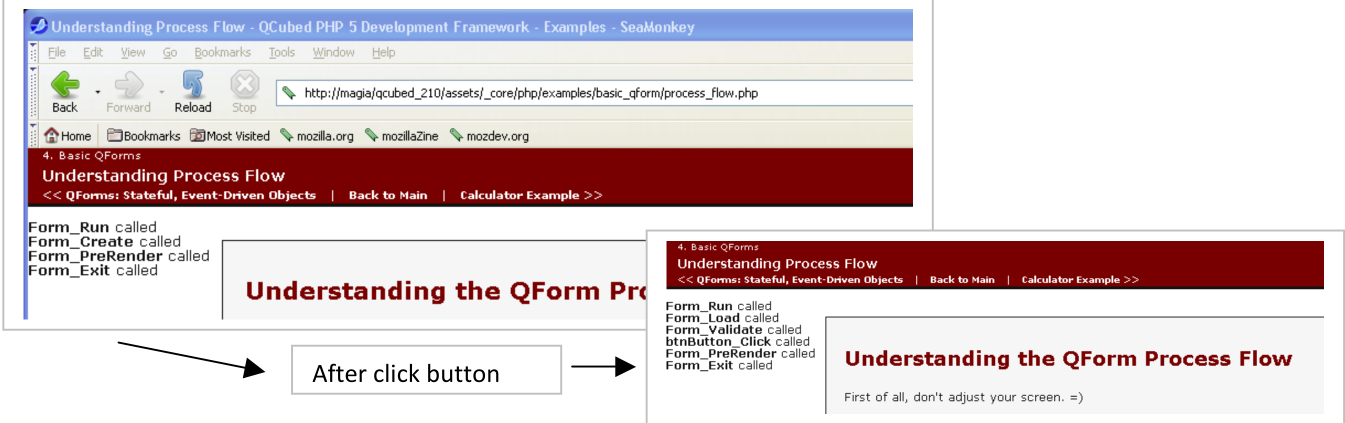
In this example, we have an $intCounter defined in the form. And basically, whenever you click on the button, we will increment $intCounter by one.

Note that the HTML template file is displaying $intCounter directly via a standard PHP print statement.

Also note that session variables, cookies, etc. are not being used here -- only FormState. In fact, you can get an idea if you do View Source... in your browser of the HTML on this page. You will see a bunch of cryptic letters and numbers for the Qform__FormState hidden variable.

```
36     <div class="instructions">
37         <h1 class="instruction_title">Understanding State</h1>
38         Note that when you clicked on the button, the form actually posted back to itself.  However,
39         the state of the form was remembered from one webpage view to the next.  This is known as
40         <b>FormState</b>.<br/><br/>
41
42         <b>QForm</b> objects, in fact, are stateful objects that maintain its state from one post to the next.<br/>
43         <br/>
44
45         In this example, we have an <b>$intCounter</b> defined in the form.  And basically, whenever
46         you click on the button, we will increment <b>$intCounter</b> by one.  Note that the HTML template
47         file is displaying <b>$intCounter</b> directly via a standard PHP <b>print</b> statement.<br/><br/>
48
49         Also note that session variables, cookies, etc. are <i>not</i> being used here -- only <b>FormState</b>.  In fact,
50         you can get an idea if you do <b>View Source...</b> in your browser of the HTML on this page.
51         You will see a bunch of cryptic letters and numbers for the <b>Qform__FormState</b> hidden variable.
52         Those letters and numbers actually represent the serialized version of this <b>QForm</b> object.
53     </div>
54
55     <p>The current count is: 0</p>
56
57     <p><span id="cl_ctl" ><button type="button" name="cl" id="cl" class="button"  > Click Me! </button></span></p>
58
59
60 <script type="text/javascript" src="http://ajax.googleapis.com/ajax/libs/jquery/1.7.1/jquery.min.js"></script>
61 <script type="text/javascript" src="http://ajax.googleapis.com/ajax/libs/jqueryui/1.8.18/jquery-ui.min.js"></script>
62 <script type="text/javascript" src="/qcubed_210/assets/_core/js/qcubed.js"></script>
63
64 <div style="display: none;">
65     <input type="hidden" name="Qform__FormState" id="Qform__FormState" value="1" />
66     <input type="hidden" name="Qform__FormId" id="Qform__FormId" value="ExamplesForm" />
67 </div>
68 <input type="hidden" name="Qform__FormControl" id="Qform__FormControl" value="" /><input type="hidden" name="Qform__FormEvent" id="Qform__FormEvent" value="" /><i
69 </form><script type="text/javascript">$j(document).ready(function() { qc.imageAssets = "/qcubed_210/assets/_core/images"; qc.cssAssets = "/qcubed_210/assets/_core
70                                 event.preventDefault(); qc.pB('ExamplesForm', 'cl', 'QClickEvent', '');
71                                 });
72                                 ;;qc.regCA(new Array("cl")); ; });</script>         </div>
73             <div id="footer">
74                 <div id="footerLeft"><a href="http://qcu.be/" title="Visit the QCubed Homepage"><img src="/qcubed_210/assets/_core/images/qcubed_logo_footer.png"
75                 <div id="footerRight">
76                     <div><span class="footerSmall">For more information, please visit the QCubed website at <a href="http://www.qcu.be/" class="footerLink">http:/
77                     <div><span class="footerSmall">Questions, comments, or issues can be discussed at the <a href="http://qcu.be/forum" class="footerLink">Example
78                 </div>
79             </div>
80         </div>
81         <script type="text/javascript">
82         var gaJsHost = (("https:" == document.location.protocol) ? "https://ssl." : "http://www.");
83         document.write(unescape("%3Cscript src='" + gaJsHost + "google-analytics.com/ga.js' type='text/javascript'%3E%3C/script%3E"));
84         </script>
85         <script type="text/javascript">
86         try {
87         var pageTracker = _gat._getTracker("UA-7231795-1");
88         pageTracker._trackPageview();
89         } catch(err) {}
90         </script>
91     </body>
92 </html>
```

Understanding the QForm Process Flow



First of all, don't adjust your screen. =)

The "Form_blah called" messages you see are showing up to illustrate how the **QForm** process flow works.
As we mentioned earlier, **QForm** objects are stateful, with the state persisting through all the user interactions (e.g. ServerActions, etc.). But note that **QForm** objects are also event-driven. This is why the we state that QForms is a stateful, event-driven architecture for web-based forms." On every execution of a **QForm**, the following actions happen:

1. The first thing the Form object does is internally determine if we are viewing this page fresh (e.g. not via a post back) or if we have actually posted back (e.g. via the triggering of a control's action which would post back to the server).
2. If it is posted back, then it will retrieve the form's state from the **FormState**, which is a hidden form variable containing the serialized data for the actual Form instance. It will then go through all the controls and update their values according to the user-entered data submitted via the post, itself.
3. Next, regardless if we're post back or not, the **Form_Run** method (if defined) will be triggered. Again, this will be run regardless if we're viewing the page fresh or if we've re-posted back to the page.
4. Next, if we are viewing the page fresh (e.g. not via a post back), the **Form_Create** method (if defined) will be run (**Form_Create** is typically where you would define and instantiate your various **QForm** controls). Otherwise, the **Form_Load** (if defined) will be run.
   - Next, if we're posted back because of a **QServerAction** or **QAjaxAction** that points to a specific PHP method, then the following will happen:
   - First, if the control that triggered the event has its CausesValidation property set, then the form will go through validation. The form will call Validate() on the relavent controls, and then it will call Form_Validate on itself. (More information on validation can be seen in the upcoming Calculator examples.)
   - Next, if validation runs successfully or if no validation is requested (because CausesValidation was set to false), then the PHP method that the action points to will be run.

   So in this repeat of the "Hello World" example, when you click on **btnButton**, the **btnButton_Click** method will be excuted during this step.
5. If defined, the **Form_PreRender** method will then be run.
6. The HTML include template file is included (to render out the HTML).
7. And finally, the **Form_Exit** (if defined) is run after the HTML has been completely outputted.

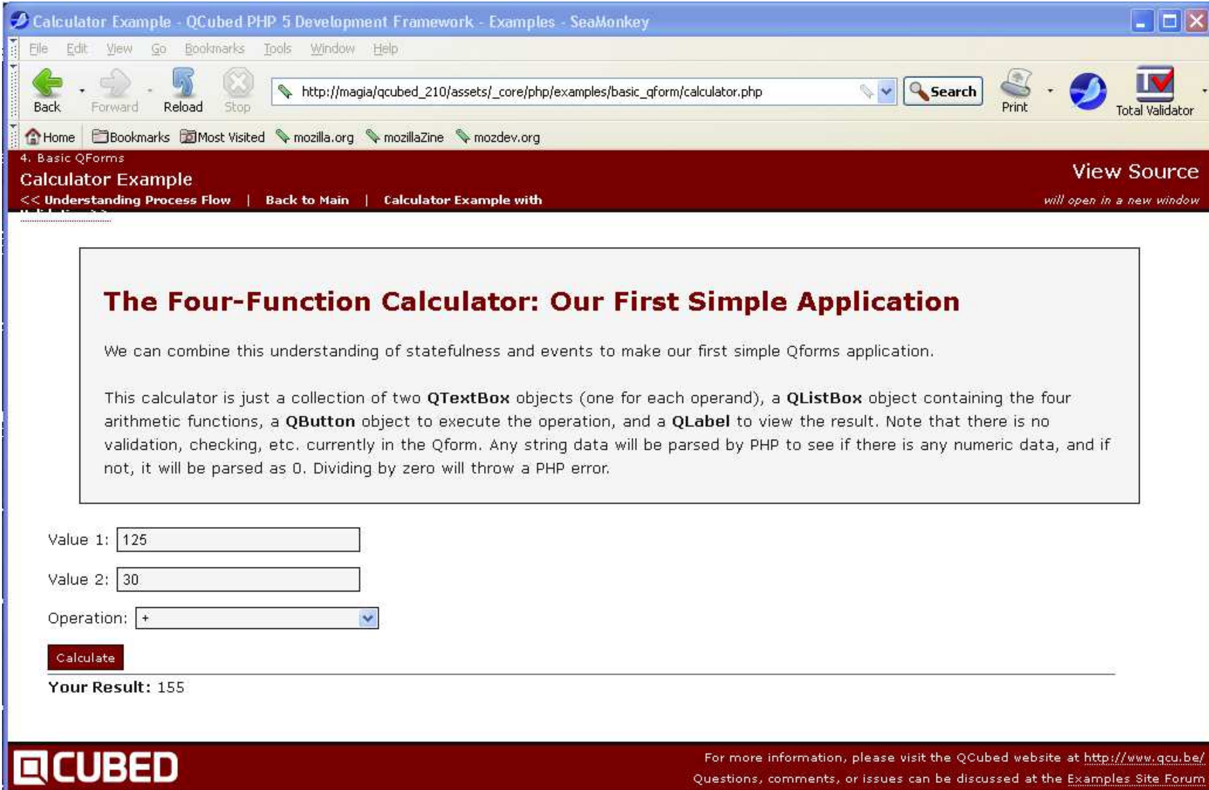
So, basically, a **QForm** can have any combination of the five following methods defined to help customize QForm and QControl processing:

- Form_Run
- Form_Load
- Form_Create
- Form_Validate
- Form_PreRender
- Form_Exit

THE FOUR-FUNCTION CALCULATOR: OUR FIRST SIMPLE APPLICATION

We can combine this understanding of statefulness and events to see and interact whit the first simple Qforms application in qcubed example pages.

This calculator is just a collection of two **QTextBox** objects (one for each operand), a **QListBox** object containing the four arithmetic functions, a **QButton** object to execute the operation, and a **QLabel** to view the result. Note that there is no validation, checking, etc. currently in the Qform. Any string data will be parsed by PHP to see if there is any numeric data, and if not, it will be parsed as 0. Dividing by zero will throw a PHP error.



You can see the code in example pages clicking on view source.

LEARNING ABOUT VALIDATION

In this example, we extend our calculator application to include Validation.

As we mentioned earlier, Qforms will go through a validation process just before it executes any Server-based actions, if needed. If the Control that triggers the ServerAction has its **CausesValidation** property set to "true", then before executing the ServerAction, the Form will go through every visible control in the entire Form and call **Validate()**. Only after ensuring that every control is valid, will the Form go ahead and execute the assigned ServerAction. Otherwise, every Control that had its **Validate()** fail will have its ValidationError property set with the appropriate error message.

What the validation checks for is dependent on the control you are using. In general, QControls that have their **Required** property set to "true" will check to ensure that data was at least entered or selected. Some controls have additional rules. For example, we'll use **QIntegerTextBox** here to have Qforms ensure that the data entered in our two textboxes are valid integers.

So we will utilize the Qforms validation in our application by doing the following:

Set our **btnCalculate** button's **CausesValidation** property to true

Use **QIntegerTextBox** classes

For those textboxes, we will use **RenderWithError()** instead of **Render()** in the HTML template code. This is because **Render()** only renders the control, itself, with no other markers or placeholders. **RenderWithError()** will be sure to render any error/warning messages for that control if needed.

Lastly, we will add our first "business rule": ensure that the user does not divide by 0. This rule will be implemented as an **if** statement in the **Form_Validate** method.

For more advanced users, note that **CausesValidation** can also be set to **QCausesValidation::SiblingsAndChildren** or **QCausesValidation::SiblingsOnly**. This functionality is geared for developers who are creating more complex **QForms** with child controls (either dynamically created, via custom composite controls, custom **QPanels**, etc.), and allows for more finely-tuned direction as to specify a specific subset of controls that should be validated, instead of validating against all controls on the form.

SiblingsAndChildren specifies to validate all sibling controls and their children of the control that is triggering the action, while **SiblingsOnly** specifies to validate the triggering control's siblings, only.

Go to example page to see this code in action and view code itself with detailed code and function explanation:

`calculator_2.php`

CUSTOM RENDERERS AND CONTROL PROPERTIES

In our final Calculator example, we show how you can use custom renderers to affect layout, as well as use control properties to change the appearance of your QControls.

The Qcubed distribution includes a sample custom renderer, **RenderWithName**, which is defined in your QControl custom class (which is at `/includes/qform/QControl.inc`). We'll use this **RenderWithName** for our calculator's textboxes and listbox. We've also made sure to assign **Name** properties to these QControls.

Note how "Value 1" and "Value 2" are in all caps and boldfaced, while "Operation" is not. This is because the textboxes are set to **Required** while the listbox is not. And the sample **RenderWithName** method has code which will boldface/allcaps the names of any required controls.

We've also made some changes to the styling and such to the various controls. Note that you can programmatically make these changes in our form definition (in **Form_Create**), and you can also make these changes as "Attribute Overrides" in the HTML template itself (see the "Other Tidbits" section for more information on **Attribute Overriding**). And finally, in our HTML template, we are now using the **RenderWithName** calls. Because of that, we no longer need to hard code the "Value 1" and "Value 2" HTML in the template.

Go to example page to see this code in action and view code itself with detailed code and function explanation:

calculator_3.php

Basic AJAX in QForms - A look at how to AJAX-enable your QForms

This example revisits our original calculator example to show how you can easily change a postback-based form and interactions into AJAX-postback based ones.

Whereas before, we executed a **QServerAction** on the button's click, we have now changed that to a **QAjaxAction**. Everything else remains the same, and now, we've created an AJAX-based calculator.

The result is the exact same interaction, but now performed Asynchronously via AJAX.

Note that even things like validation messages, etc., will appear via AJAX and without a page refresh.

Go to example page to see this code in action and view code itself with detailed code and function explanation:

basic_ajax/calculator_2.php

More About Events and Actions –

Here we go more in depth at the capabilities of the QEvent and QAction libraries

Combining Multiple Actions on Events

We can combine mulitple actions together for events, and we can also use the same set of actions for on multiple events or controls.

In this example, we have a listbox, and we allow the user to dynamically add items to that listbox. On submitting, we want to perform the following actions:

- Disable the Listbox (via Javascript)
- Disable the Textbox (via Javascript)
- Disable the Button (via Javascript)
- Make an AJAX call to the PHP method **AddListItem**

The PHP method **AddListItem** will then proceed to add the item into the listbox, and re-enable all the controls that were disabled.

Note that what we are doing is combining multiple actions together into an action array (e.g. **QAction[]**). Also note that this action array is defined on two different controls: the button (as a **QClickEvent**) and the textbox (as a **QEnterKeyEvent**).

Also note that we also add a **QTerminateAction** action to the textbox in response to the **QEnterKeyEvent**. The reason for this is that on some browsers, hitting the enter key in a textbox would cause the form to do a traditional form.submit() call. Given the way Qforms operates with named actions, and especially given the fact that this Qform is using AJAX-based actions, we do *not* want the browser to be haphazardly performing submits.

Finally, while this example uses **QAjaxAction** to make that an AJAX-based call to the PHP **AddListItem** method, note that this example can just as easily have made the call to **AddListItem** via a standard **QServerAction**. The concept of combining multiple actions together and the concept of reusing an array of actions on different controls/events remain the same.

Go to example page to see this code in action and view code itself with detailed code and function explanation:

events_actions/editable_listbox.php

Making Events Conditional

Somtimes we want events to trigger conditionally. Given our editable listbox, a good example of this is that we want the submitting of the new Item to only happen if the user has typed in something in the textbox.

Basically, if the textbox is blank, no event should trigger. (You can verify this now by clicking "Add Item" without while keeping the textbox completely blank.)

Qcubed supports this by allowing all events to have optional conditions. These conditions are written as custom javascript code into the Event constructor itself.

In this example, we explicitly name the textbox's ControlId as "txtItem" so that we can write custom javascript as conditionals to the button's **QClickEvent** and the textbox's **QEnterKeyEvent**.

Go to example page to see this code in action and view code itself with detailed code and function explanation:

events_actions/editable_listbox_2.php

Triggering Events after a Delay

Sometimes, you may want events to trigger their assigned actions after a delay. A good example of this here is the **QKeyPressEvent** we added below. As the user enters in data into the textbox, we make an AJAX call to update the label. However, in order to make the system a bit more usable and streamlined, we have added a half-second (500 ms) delay on the **QKeyPressEvent**, so that we are not making too many AJAX calls as the user is still entering in data.

Basically, this allows the action to be triggered only after the user is done typing in the data.

Note that we maybe could have used a QChangeEvent on the textbox to achieve a similar effect. But realize that QChangeEvent (which utilizes a javascript **onchange** event handler) will only be triggered after the control *loses focus* and has been changed -- it won't be triggered purely by the fact that the text in the textbox has been changed.

Go to example page to see this code in action and view code itself with detailed code and function explanation:

events_actions/delayed.php

Triggering Arbitrary JavaScript, Alerts and Confirms

Qcubed includes several commonly used Javascript-based actions:

- **QAlertAction** - to display a javascript "alert" type of dialog box
- **QConfirmAction** - to display a javascript "confirm" type of dialog box, and execute following optional actions if the user hits "Ok"
- **QJavaScriptAction** - to run any arbitrary javascript command(s)

The example below shows three different **QButton** controls which use all three of these action types.

Specifically for the **QJavaScriptAction**, we've defined a simple **SomeArbitraryJavaScript()** javascript function on the page itself, so that the button has some javascript to perform.

If you are interested in more advanced and flexible types of confirmation or prompts, take a look at the plugin that offers pre-built QDialogBox subclasses.

Go to example page to see this code in action and view code itself with detailed code and function explanation:

events_actions/javascript_alerts.php

Other Client-Side Action Types

Below is a sampling of just *some* of the other **QAction** types that are available to you as part of the core Qcubed distribution.

Notice that all of these **QActions** simply render out javascript to perform the action, so the interaction the user experience is completely done on the client-side (e.g. no server/ajax calls here).

View the code for the details, and for more information or for a listing of *all* the **QActions** and **QEvents**, please see the **Documentation** section of the Qcubed website.

Go to example page to see this code in action and view code itself with detailed code and function explanation:

events_actions/other_actions.php

Paginated Controls - The QDataGrid and QDataRepeater controls

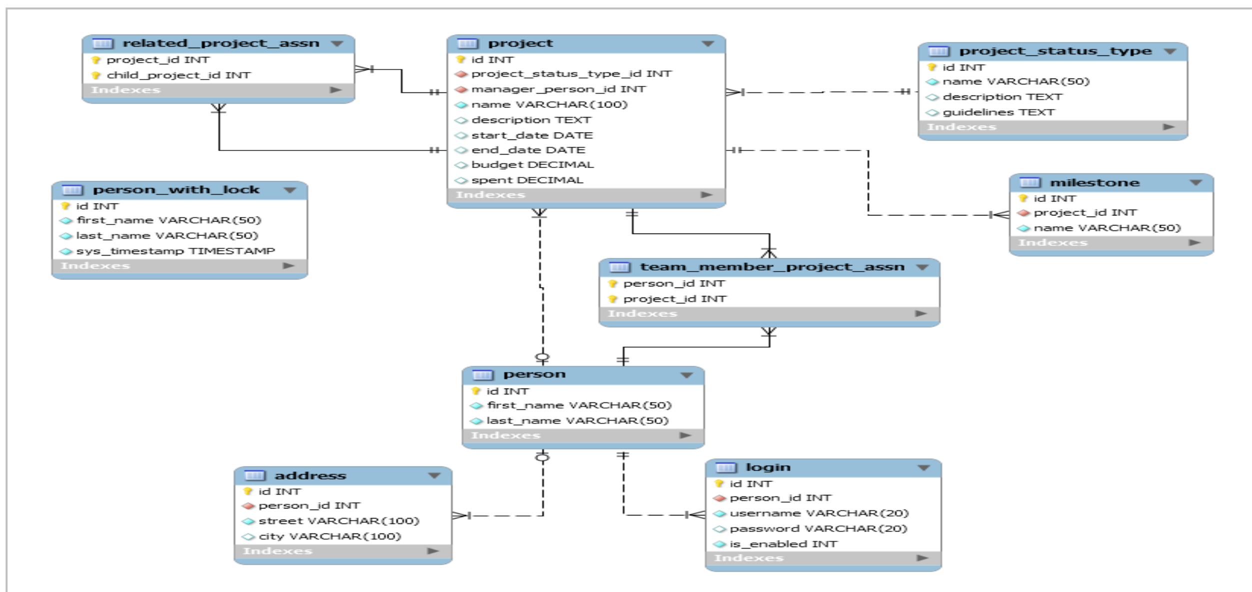
To go ahead and have next examples working we need the **Examples Site Database**. This database (which consists of six tables and some preloaded sample data) is included in the Examples Site directories (D:\wamp\www\qcubed_210\assets_core\php\examples\mysql_innodb.sql).

The Examples Site Database

examples/code_generator/intro.php

Before learning about the Code Generator, it might be good to first get acquainted with the data model which the Code Generator will be generating from.

Click on the "View Source" link in the upper righthand corner to view the **mysql_innodb.sql** to examine the data model in script form, or you can view an ER diagram of the data model below.



If you have not installed this **Examples Site Database** on your MySQL server, you might want to do that now.

After installing the tables (please leave untouched the db timetrack), you must also remember to change our db section of `includes\configuration.inc.php` to reference the created database and go to index_install (you can remember here that we renamed original installed index.php in previous chapter to index_install.php) after intallation to **code generate** the corresponding objects *before* trying to any of the further code generation examples.

The code generated will be added to that one we generated in previous chapter_07, so, if you do not want be confused, you can just clear previous generated code or leave it knowing this fact(*table person is new and different from user and project and this will be source of confusion if you tray to login and interact with code of previous chapter !!!!).*

A good exercise will be join the logic and table structure and let previous example work with this db. We willl se this later.

At momente we saved all in a complete zip for chapter 07 (`qcubed_210_chapt_07.zip`) and started with the new generated code and a qcubed reinstallation retaining configuration.inc and save, for later use, our revised styles.css we used in prevoious chapter.

Note that there is also a SQL Server version of this database script called **sql_server.sql**.

In the script, we have six tables defined. The bulk of our examples will focus on the main three tables of the database:

- **login**
- **person**
- **project**

The **team_member_project_assn** table handles the many-to-many relationship between **person** and **project**. The **project_status_type** table is a **Type Table** which will be discussed in the example for **Type Tables**. Finally the **person_with_lock** table is specifically used by the example for **Optimistic Locking**.

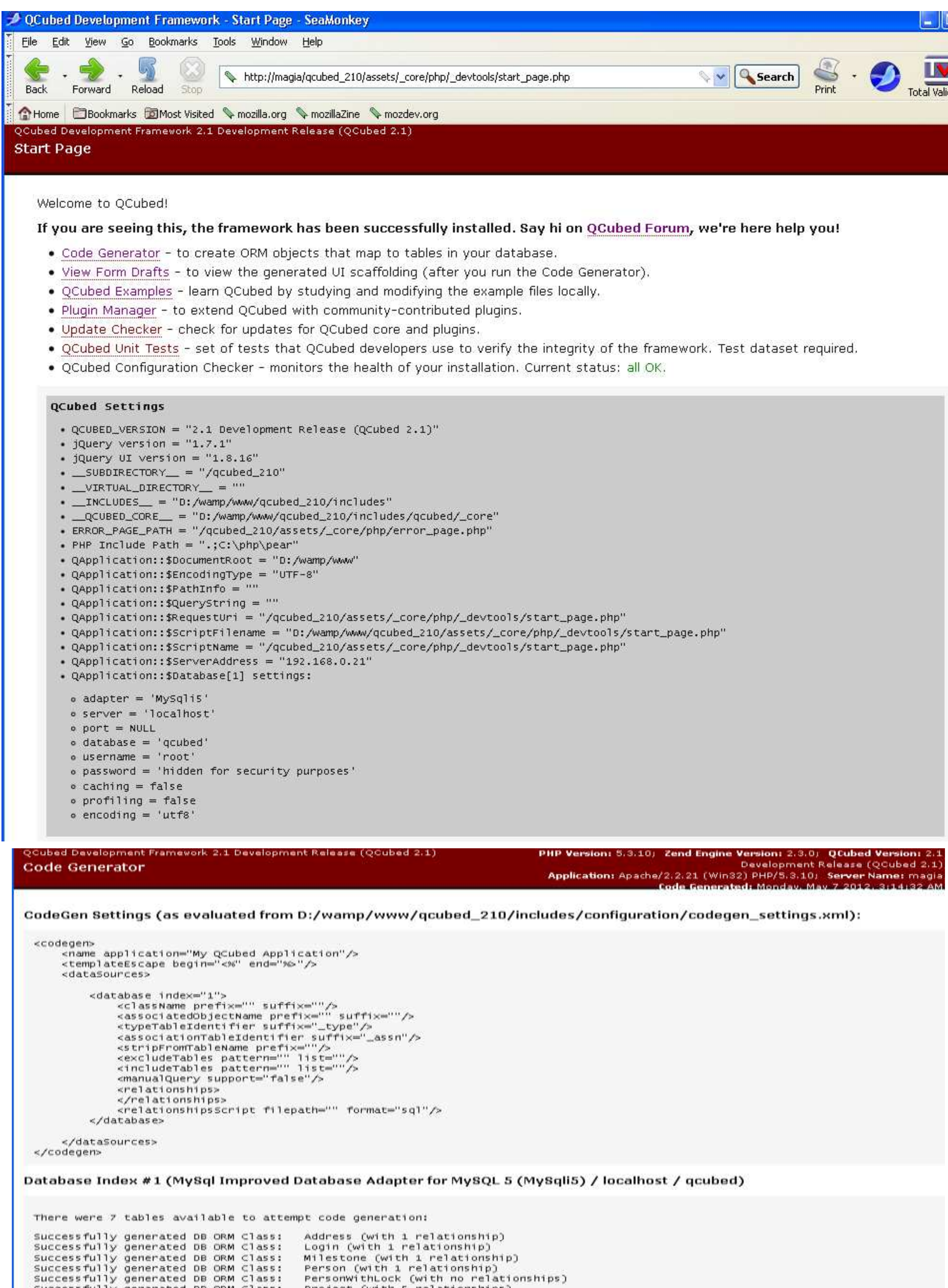
Now, is time to create new db (name qcubed) loading table (you can use import sql),

change`includes\configuration.inc.php` to point to this new db

```
define('DB_CONNECTION_1', serialize(array(
   'adapter' => 'MySqli5',
   'server' => 'localhost',
   'port' => null,
   'database' => 'qcubed',
   'username' => 'root',
   'password' => '',
   'profiling' => false,
   'encoding' => 'utf8' )));
```

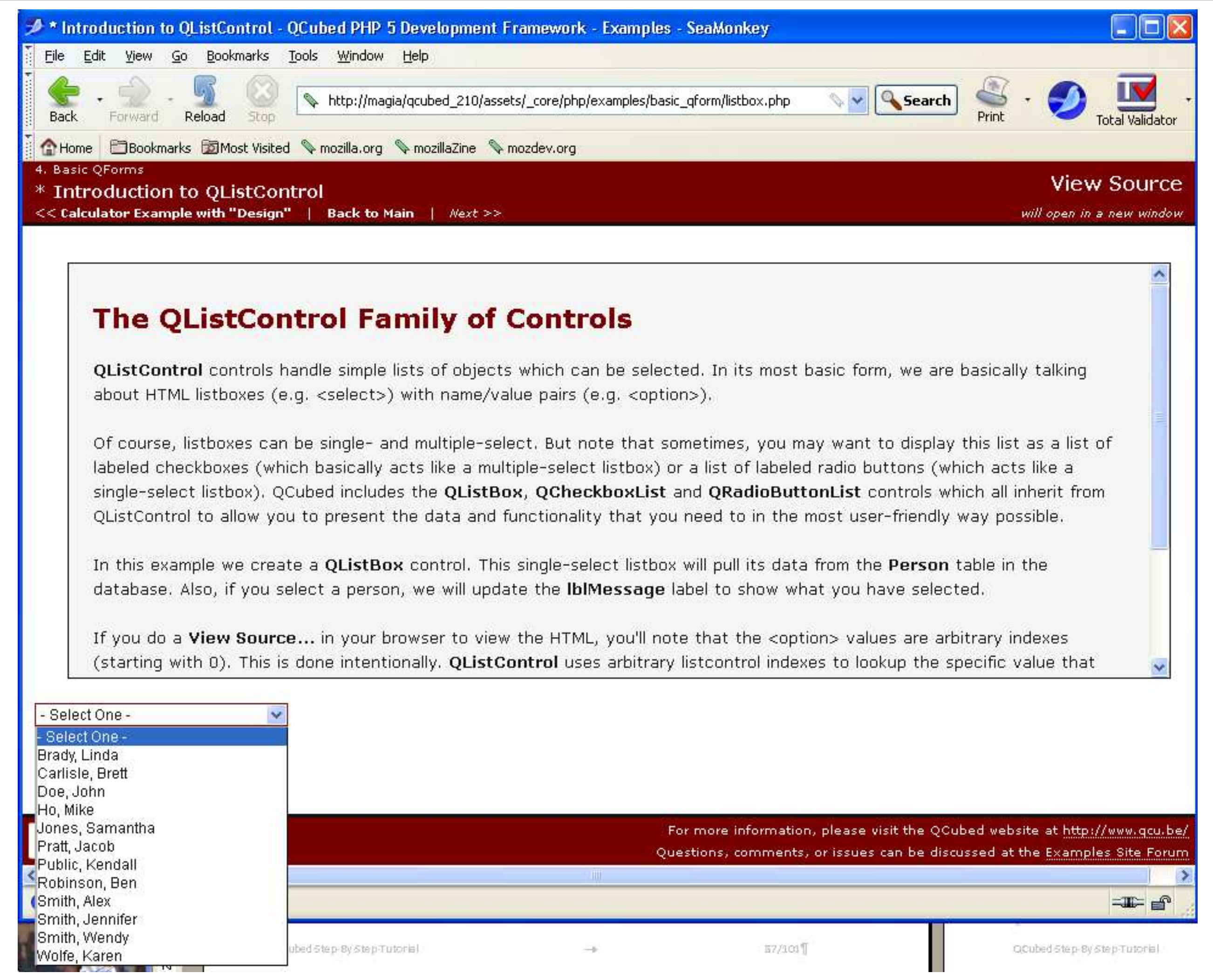
If not a Qcubed reinstallation clear old generated code (remember to retain index.php and index.tpl.php in drafts and drafts/dashbard) , call index_install.php (or index.php if new installation) and regen.

The result wil be similar to this one:



Now we can load example pages * (requiring db example related table) and see that them now work.

basic_qform/listbox.php



Paginated Controls - The QDataGrid and QDataRepeater controls

An Introduction to the QDataGrid Class

The **QDataGrid** control is used to present a collection of objects or data in a grid-based (e.g. <table>) format. All **QDataGrid** objects take in a **DataSource**, which can be an array of anything (or in our example, an array of Person objects).

In defining a **QDataGrid**, you must define a new **QDataGridColumn** for each column in your table. For each **QDataGridColumn** you can specify its name and how it should be rendered. The HTML definition in your **QDataGridColumn** will be rendered directly into your HTML output. Inside your HTML definition, you can also specify PHP commands, methods, function calls and/or variables which can be used to output item-specific data.

Calls to PHP can be made by using <?= and ?> tags (see this example's code for more information). Note that these PHP short tags are being used by Qcubed *internally* as delimiters on when the PHP engine should be used. **QDataGrid** (and Qcubed in general, for that matter) offers full support of PHP installations with **php_short_tags** set to off.

Finally, the **QDataGrid**'s style is fully customizable, at both the column level and the row level. You can specify specific column style attributes (e.g. the last name should be in bold), and you can specify row attributes for all rows, just the header, and just alternating rows.

Go to example page to see this code in action and view code itself with detailed code and function explanation:

datagrid/intro.php

The QDataGrid Variables -- $_ITEM, $_COLUMN, $_CONTROL and $_FORM

As you may have noticed in the first example, we make use of the $_ITEM variable when we render each row's column. There are in fact three special variables used by the QDataGrid: **$_ITEM**, **$_COLUMN**, **$_CONTROL** and **$_FORM**.

$_ITEM represents a specific row's instance of the array of items you are iterating through. So in our example, the **DataSource** is an array of **Person** objects. Therefore, **$_ITEM** is the specific **Person** object for the row that we are rendering. **$_COLUMN** is the QDataGridColumn, **$_CONTROL** is the QDataGrid itself and **$_FORM** is the QForm itself.

So in our example, the first column shows the "Row Number", which is basically just the **CurrentRowIndex** property of the **QDataGrid** (e.g. **$_CONTROL**). And the last column's "Full Name" is rendered by the **DisplayFullName** method we have defined in our **ExampleForm** (e.g. **$_FORM**). Note that the **DisplayFullName** takes in a **Person** object. Subsequently, in our HTML defintion, we make the call to **$_FORM->DisplayFullName** passing in **$_ITEM**.

Finally, note that **DisplayFullName** is declared as a **Public** method. This is because **DisplayFullName** is actually called by the **QDataGrid**, which only has the rights to call **Public** methods in your **ExampleForm** class.

Go to example page to see this code in action and view code itself with detailed code and function explanation:

datagrid/variables.php

Sorting a QDataGrid by Columns

In this example we show how to make the datagrid sortable by individual columns.

For each column, we add the properties **OrderByClause** and **ReverseOrderByClause** (it is possible to also just define **OrderByClause**, and to leave **ReverseOrderByClause** undefined). The **QQ Clause** you specify is given back to you when you call the **OrderByClause** property on the **QDataGrid** itself.

So what you do is you specify the **QQ OrderBy Clause** that you would want run for each column. Then you pass the this clause to your class's **LoadAll** or **LoadArrayByXXX** method as one of the optional **QQ Clause** parameters. Note that all Qcubed code generated **LoadAll** and **LoadArrayByXXX** methods take in an optional **$objOptionalClauses** parameter which conveniently uses the clause returned by the **QDataGrid**'s **OrderByClause** method.

Convenient how they end up working together, isn't it? =)

Go to example page to see this code in action and view code itself with detailed code and function explanation:

datagrid/sorting.php

Adding Pagination to Your QDataGrid

Now, we will add pagination to our datagrid.

In order to enable pagination, we need to define a **QPaginator** object and assign it to the **QDataGrid**. Because the **QPaginator** will be rendered by the **QDataGrid** (instead of being rendered on the form via a **$this->objPaginator->Render()** call), we will set the **QDataGrid** as the **QPaginator**'s parent in the **QPaginator** constructor call.

In the locally defined **dtgPersons_Bind** method, in addition to setting the datagrid's **DataSource**, we also give the datagrid the **TotalItemCount** (via a **Person::CountAll** call). And finally, when we make the **Person::LoadAll** call, we make sure to pass in the datagrid's **LimitClause**, which will pass the paging information into our **LoadAll** call to only retrieve the items on the page we are currently viewing.

Go to example page to see this code in action and view code itself with detailed code and function explanation:

datagrid/pagination.php

Enabling AJAX-based Sorting and Pagination

In this example, we modify our sortable and paginated **QDataGrid** to now perform AJAX-based sorting and pagination.

We literally just add *one line* of code to enable AJAX.

By setting **UseAjax** to **true**, the sorting and pagiantion features will now execute via AJAX. Try it out, and notice how paging and resorting doesn't involve the browser to do a full page refresh.

Go to example page to see this code in action and view code itself with detailed code and function explanation:

datagrid/ajax.php

SIMPLE ERGONOMIC TIP ON DATAGRID INTERFACE

You can see that pagination info are at right position of page? So, sometime we must horizontal scroll to join those info.

I suggest a simple modification to put those info in more ergonomic position after result info.

You can refer to my suggestion on Qcubed forum simple-ergonomic-tip-datagrid-interface .

In that tip I suggest some modification to QDataGridBase.class.php (located in .\includes\qcodo_core\qform\) and to my styles.css (located in .\assets\css\)

The two modified files are in this chapter zip as substitution to the original present in previous chapter zip (the original version are saved as *.original).

As Style, I choose also to use 10px as font dimension. Simple trick, but very appreciated by my users.

`QDataGridBase.class.php` compare screen

Windows XP Pro_2009 - [Ctrl-Alt-F1] - VMware Workstation

File Power Settings Devices View Help

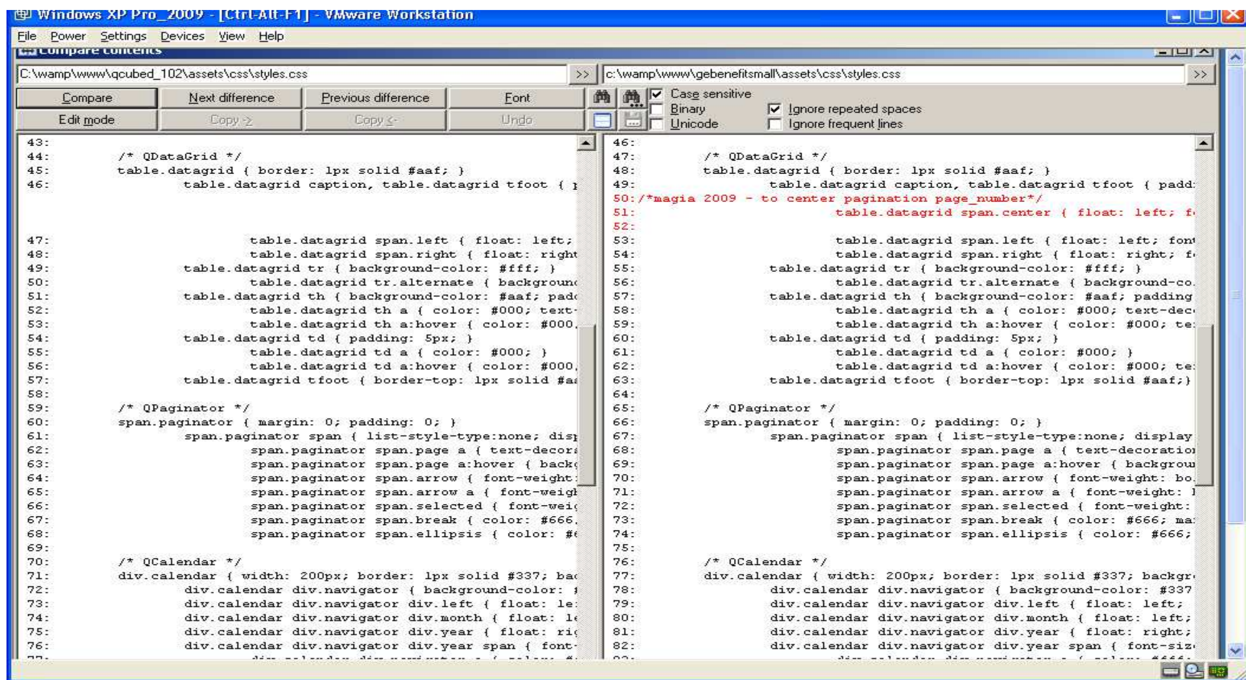
```
477:                        }
478:                }
479:        }
480:
481:        protected function GetPaginatorRowHtml($objPagi
482:                $strToReturn = "  <span class=\"right\">

483:                $strToReturn .= $objPaginator->Render(fa
484:                $strToReturn .= "</span>\r\n  <span clas

485:                if ($this->TotalItemCount > 0) {
486:                        $intStart = (($this->PageNumber
487:                        $intEnd = $intStart + count($th
488:                        $strToReturn .= sprintf($this->s
489:                                $this->strNounPlural,
490:                                $intStart,
491:                                $intEnd,
492:                                $this->TotalItemCount);
493:                } else {
494:                        $intCount = count($this->objData
495:                        if ($intCount == 0)
496:                                $strToReturn .= sprintf
497:                        else if ($intCount == 1)
498:                                $strToReturn .= sprintf
499:                        else
500:                                $strToReturn .= sprintf
501:                }
502:

503:                $strToReturn .= "</span>\r\n";
504:
505:                return $strToReturn;
506:        }
507:
```

```
477:                        }
478:                }
479:        }
480:
481:        protected function GetPaginatorRowHtml($objPaginato
482:/* magia 2009 to have in left side paginator */
483:                // this two line moved at the end and appende
484:
485:                /*        $strToReturn = "  <span class=\"rig
486:                  $strToReturn .= $objPaginator->Render(false
487:               */
488:                  // remember to remove  . from $strToReturn
489:                  $strToReturn = "</span>\r\n  <span class=\".
490:                  if ($this->TotalItemCount > 0) {
491:                          $intStart = (($this->PageNumber - 1
492:                          $intEnd = $intStart + count($this->
493:                          $strToReturn .= sprintf($this->strL
494:                                  $this->strNounPlural,
495:                                  $intStart,
496:                                  $intEnd,
497:                                  $this->TotalItemCount);
498:                  } else {
499:                          $intCount = count($this->objDataSou
500:                          if ($intCount == 0)
501:                                  $strToReturn .= sprintf($th
502:                          else if ($intCount == 1)
503:                                  $strToReturn .= sprintf($th
504:                          else
505:                                  $strToReturn .= sprintf($th
506:                  }
507:
508:// magia 2009 cont
509:                  // moved from start of function and appende
510:// magia 2009 added translation
511:                  $tempGotopage=QApplication::Translate('<b>G
512:                  $strToReturn .= "   ".$tempG
513:                  $strToReturn .= $objPaginator->Render(false
514:                  $strToReturn .= "</span>\r\n";
515:
516:                  return $strToReturn;
517:          }
518:
```

Patched function GetPaginatorRowHtml in `QDataGridBase.class.php`

```
    protected function GetPaginatorRowHtml($objPaginator) {
 /* patched function - */
 /* magia 2009 to have in left side paginator
 * as per http://qcu.be/content/simple-ergonomic-tip-datagrid-interface suggestion
 *
 * require also this line in styles.css
 *   to be added
 *          table.datagrid span.center { float: left; font-size: 10px; display: block; }
 *   before
 *          table.datagrid span.left { float: left; font-size: 10px; display: block; }
 */


//        $strToReturn = "  <span class=\"right\">";
//        $strToReturn .= $objPaginator->Render(false);
//        $strToReturn .= "</span>\r\n  <span class=\"left\">";
// so we need to start a new $strToReturn
       $strToReturn = "</span>\r\n  <span class=\"left\">";

       if ($this->TotalItemCount > 0) {
          $intStart = (($this->PageNumber - 1) * $this->ItemsPerPage) + 1;
          $intEnd = $intStart + count($this->DataSource) - 1;
          $strToReturn .= sprintf($this->strLabelForPaginated,
             $this->strNounPlural,
             $intStart,
             $intEnd,
             $this->TotalItemCount);
       } else {
          $intCount = count($this->objDataSource);
          if ($intCount == 0)
             $strToReturn .= sprintf($this->strLabelForNoneFound, $this->strNounPlural);
          else if ($intCount == 1)
             $strToReturn .= sprintf($this->strLabelForOneFound, $this->strNoun);
          else
             $strToReturn    .=    sprintf($this->strLabelForMultipleFound,    $intCount,    $this-
>strNounPlural);
       }

       // magia 2009 cont
// moved from start of function and appended
// magia 2009 added translation
// label Goto page: can be already present if we installed the plugin QPagePaginated
               $testgotopage = $objPaginator->Render(false);
          $tempGotopage =".";
          if (strstr($testgotopage, '<b>Go to page:</b>') == false){
               $tempGotopage=QApplication::Translate('<b>Go to page:</b>');
               }
          $strToReturn       .=       "   ".$tempGotopage."        </span><span
class=\"center\">";
        $strToReturn .= $testgotopage;

       $strToReturn .= "</span>\r\n";

       return $strToReturn;
    }
```

`styles.css` compare screen



Patched styles.css;

```
/* QDataGrid */
table.datagrid { border: 1px solid #111; width:100%; overflow:auto; }
 table.datagrid caption, table.datagrid tfoot { padding-bottom: 4px; overflow: auto; }
          /*  added  by  Magia  2012  as  per  http://qcu.be/content/simple-ergonomic-tip-datagrid-interface suggestion         */
          /* to center pagination info instead of having it out of page position at right */
          /*  is  required  also  a  modification  to  function  GetPaginatorRowHtml  in QDataGridBase.class.php  */
          table.datagrid span.left { float: left; font-size: 10px; display: block; }
    table.datagrid span.left { float: left; font-size: 10px; display: block; }
    table.datagrid span.right { float: right; font-size: 10px; display: block; }
 table.datagrid tr { background-color: #fff; }
    table.datagrid tr.alternate { background-color: #EFEEEE; }
 table.datagrid  th  {  background-color:  #111;  padding:  3px  5px  3px  5px;  text-align:  left; color:#FFF;}
    table.datagrid th a { color: #FFF; text-decoration: none; }
    table.datagrid th a:hover { color: #FFF; text-decoration: underline;}
 table.datagrid td { padding: 5px; }
 table.datagrid tfoot { border-top: 1px solid #111;}


/* QPaginator */
```

Go to example page to see this code in action and view code itself with detailed code and function explanation:

datagrid/pagination.php

Advanced Controls Manipulation -

Now we skip some examples and go directly to example we used as base for our magic index.php,panel_drafts.php and panel_drafts.tpl.php in chapter 07 to interact with table using *directory_table* related scripts derived from panels.

Handling "Multiple QForms" on the Same Page

Qcubed only allows each front-end "web page" to only have a maximum of one **QForm** class per page. Because of the many issues of managing and maintaining formstate across multiple **QForms**, Qcubed simply does not allow for the ability to have multiple **QForms** per page.

However, as the development of a Qcubed application matures, developers may find themselves wishing for this ability:

As **QForms** are initially developed for simple, single-step tasks (e.g. "Post a Comment", "Edit a Project's Name", etc.), developers may want to be able to combine these simpler QForms together onto a single, larger, more cohesive QForm, utilizing AJAX to provide for a more "Single-Page Web Application" type of architecture.

Moreover, developers may end up with a library of these **QForms** that they would want to reuse in multiple locations, thus allowing for a much better, more modularized codebase.

Fortunately, the **QPanel** control was specifically designed to provide this kind of "Multiple **QForm**" functionality. In the example below, we create a couple of custom **QPanels** to help with the viewing and editing of a Project and its team members. The comments in each of these custom controls explain how a custom **QPanel** provides similar functionality to an independent, stand-alone **QForm**, but also details the small differences in how the certain events need to be coded.

Next, to illustrate this point further we create a **PersonEditPanel**, which is based on the code generated **PersonEditFormBase** class.

Finally, we use a few **QAjaxActions** and **QAjaxControlActions** to tie them all together into a single-page web application.

Go to example page to see this code in action and view code itself with detailed code and function explanation:

multiple_qform/intro.php

Let our timetrack code survive on new DB created for examples

Login.php modification

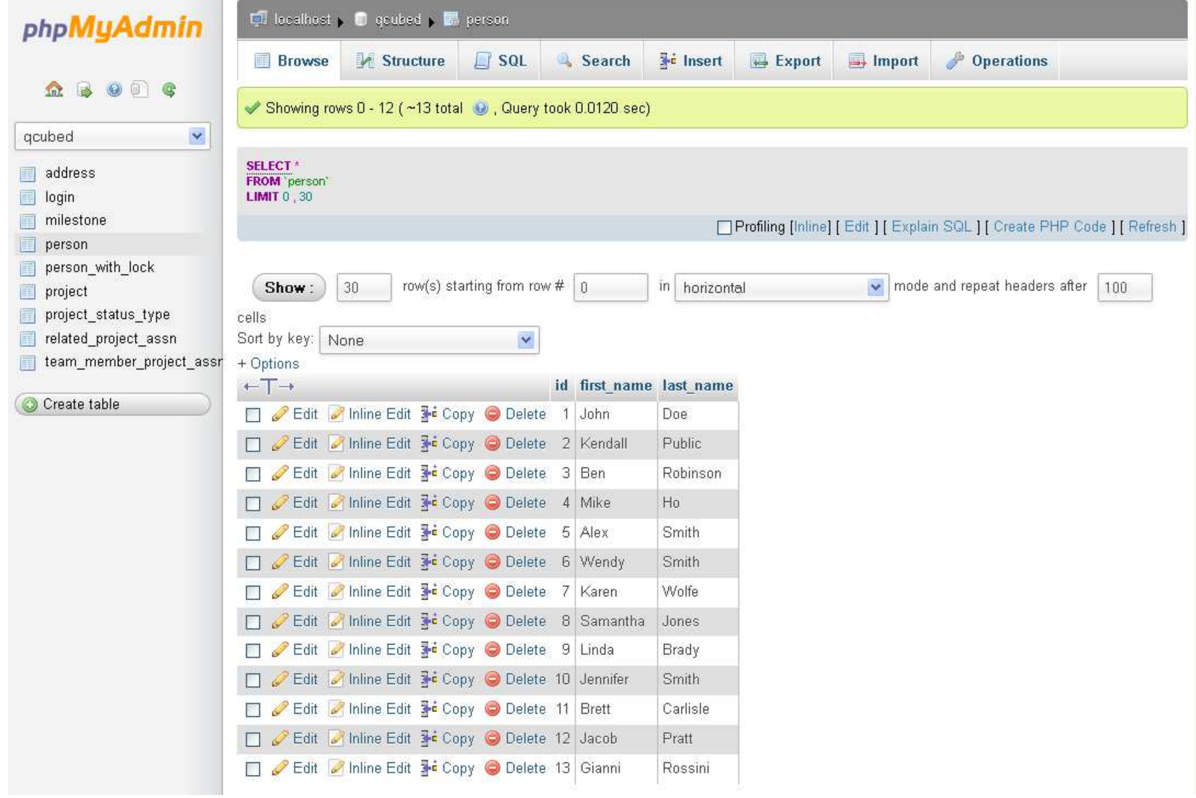
In DB related to example the table involved in login action with field username and password is Login (in chapter 07 the table used to login was User) so all controls related to User are renamed to Login:

To test our code populated login table(with phpmyadmin) the data row already present in User.

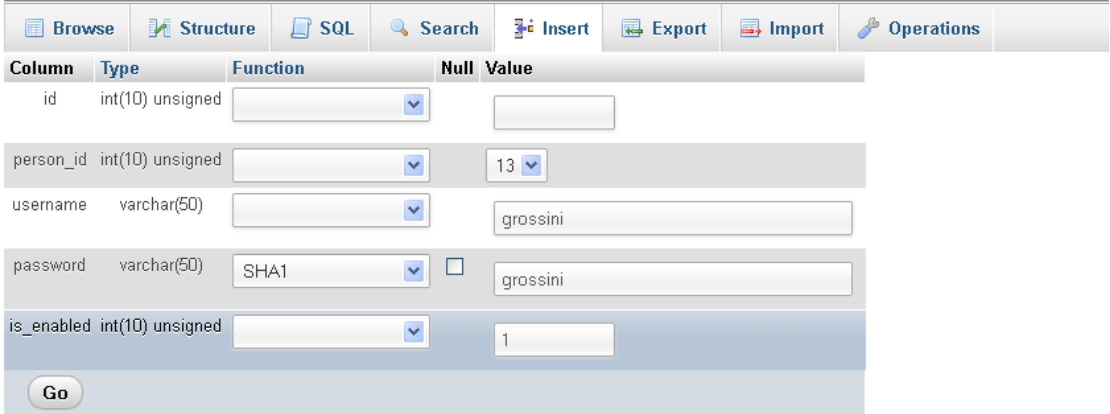
I used `grossini` as username and grossini as password. To insert password remember that this field is crypted so insert it using sha1 function.

You have already inserted one row for grossini in table Person (ID is 13).

Action on Person table



Action on Login table



Copy from Chapter 7 login.php, login.tpl.php,protected.inc.php,footer.inc.php,header.inc.php,menu.inc.php and rename index.php to index.install.php if you did not already.

In login.php change all instance of $mctUser to $mctLogin;

Change the line from

```
$this->mctLogin = UserMetaControl::CreateFromPathInfo($this);
```

To:

```
$this->mctLogin = LoginMetaControl::CreateFromPathInfo($this);
```

Change in function btnLogin_Click reference to User in reference to Login from

```
$objUser = User::LoadByUsername($this->txtUsername->Text);
```

to:

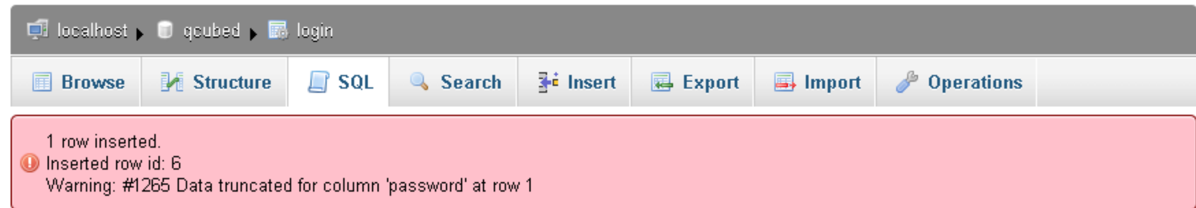
```
$objUser = Login::LoadByUsername($this->txtUsername->Text);
```

and now you can try to login.

Why our login replay “Unknown user or password” to a really correct login info?

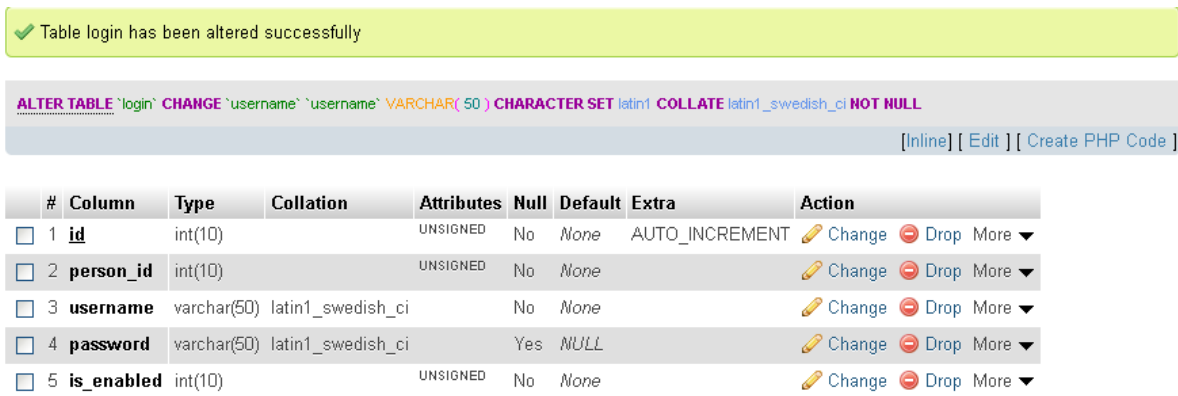
You can investigate, but I short your debug effort with the solution.

When you inserted grossini in login table mysql returned this warning message:



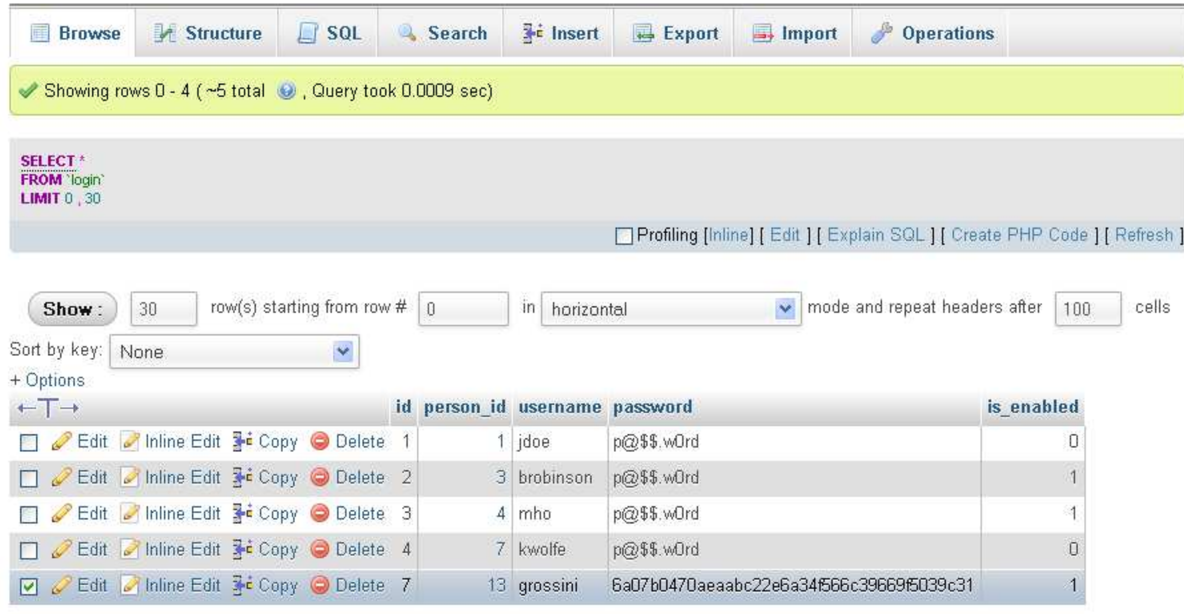
Why this warning message??

Before you broke your finger on keyboard for some obscure error consider that SHA1 mysql result is 40 chars so change on field dimension is required (50). We change also username whit this action:



#	Column	Type	Collation	Attributes	Null	Default	Extra	Action
1	id	int(10)		UNSIGNED	No	None	AUTO_INCREMENT	Change Drop More
2	person_id	int(10)		UNSIGNED	No	None		Change Drop More
3	username	varchar(50)	latin1_swedish_ci		No	None		Change Drop More
4	password	varchar(50)	latin1_swedish_ci		Yes	NULL		Change Drop More
5	is_enabled	int(10)		UNSIGNED	No	None		Change Drop More

And now result is (no warning error now):



Now your login will be ok (no unknown user or password message) but your redirection to succesfull page nedd some other work.

Also is time to regen (use index_install.php) to update object and class related to username and password field length modification.

Let we see at protected.inc.php and the redundant test on $_SESSION['User'].

The second test was unsuccessfully becouse $objuser isinstanceof(Login) not of User (table not jet exists).Now we can change instanceof to Login in protected.inc.php. to have our welcome page.

We know already how to change *Login Object* message with simple modification on `Login.class.php` , from:

```
return sprintf('Login Object %s',  $this->intId);
```

to:

```
return sprintf('%s',  $this->strUsername);
```

and the expected result is

ADD APPLICATION DIRECTORY (RELATED TO NEW TABLE)

Create the following empty directory:

```
10/10/2010   23.33    <DIR>          address
10/10/2010   23.57    <DIR>          login
10/10/2010   23.34    <DIR>          milestone
10/10/2010   23.35    <DIR>          person
10/10/2010   23.36    <DIR>          project
```

Copy inside them from `\drafts\dashboard` the appropriate `ListPanel` and `EditPanel` class and tpl.

Revise the magic index.php, panel_drafts.php, panel_drafts.tpl.php and update initial control:

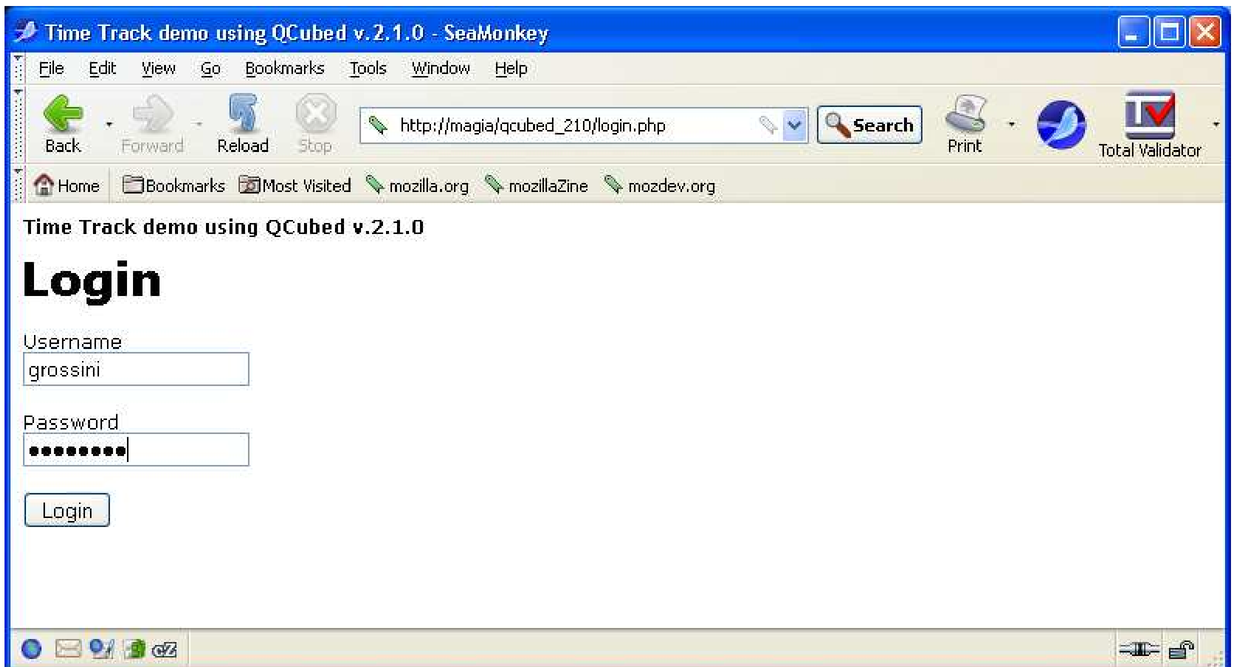
```
<?php

  // Include prepend.inc to load Qcubed
  require('../includes/configuration/prepend.inc.php');

//  forza il login se necessario
if (!isset($_SESSION['User']))
QApplication::Redirect('../login.php');

$objUser = unserialize($_SESSION['User']);

// make sure no errors occured in translation and the session's User variable is a user object
// if (!($objUser instanceof User))
   if (!($objUser instanceof Login))

QApplication::Redirect('../login.php');

if ($objUser->Username != "grossini") {
     echo "Unauthorized access";
     exit;
    }
```

Note that the authorized user is grossini .

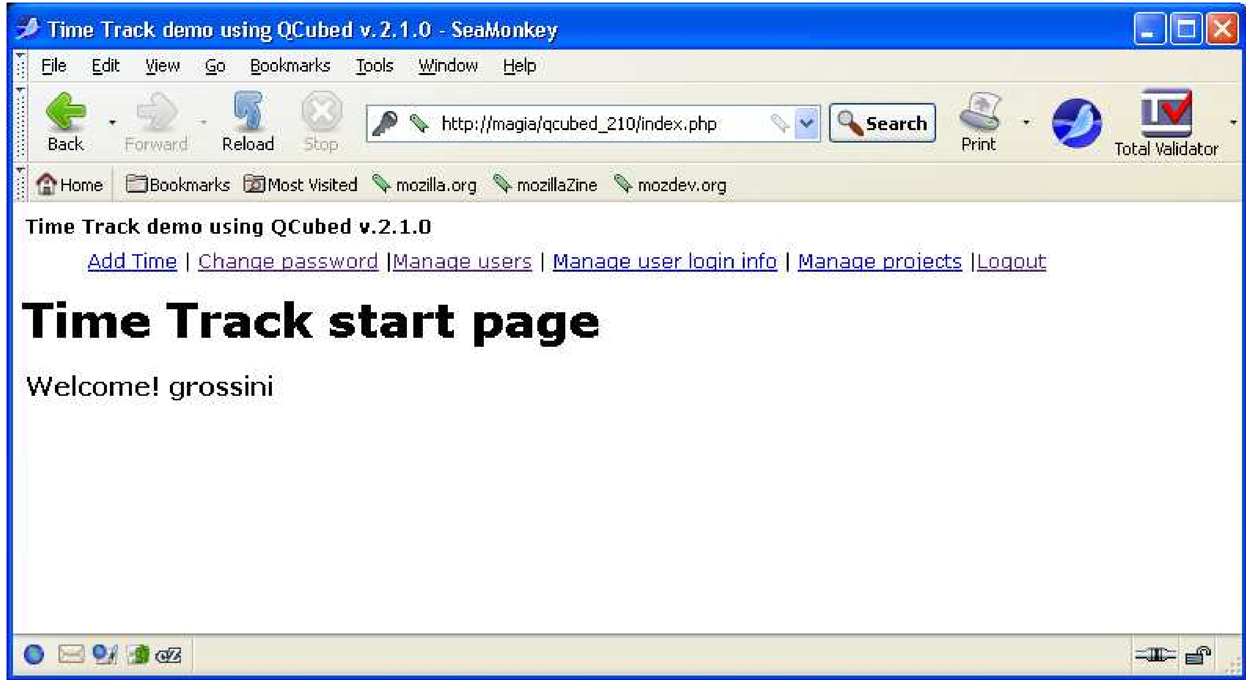
Now you can copy revised magic index.php, panel_drafts.php, panel_drafts.tpl.php in every application directory and add the links in menu.inc.php. visible only to grossini (and copy also our old logout.php, changepassword.php, logout.tpl.php, changepassword.tpl.php). I suggest also that is the right moment to use my suggested styles.css.

```
<?php
     if (isset($_SESSION['User'])){
       $objUser  =  unserialize  ($_SESSION['User']);
       if  (($objUser->Username)){
       echo '<a href="milestone/index.php">Add Time</a> |
          <a href="changepass.php">Change password</a> |';
          if  (($objUser->Username)== 'grossini'){
          echo '<a  href="person/index.php">Manage users</a>  |
              <a  href="login/index.php">Manage user login info</a>  |
                 <a  href="project/index.php">Manage  projects</a>  |';
       }
    echo '<a href="logout.php">Logout</a>';
    }
  } ?>Now we can try a new login as grossini:
```

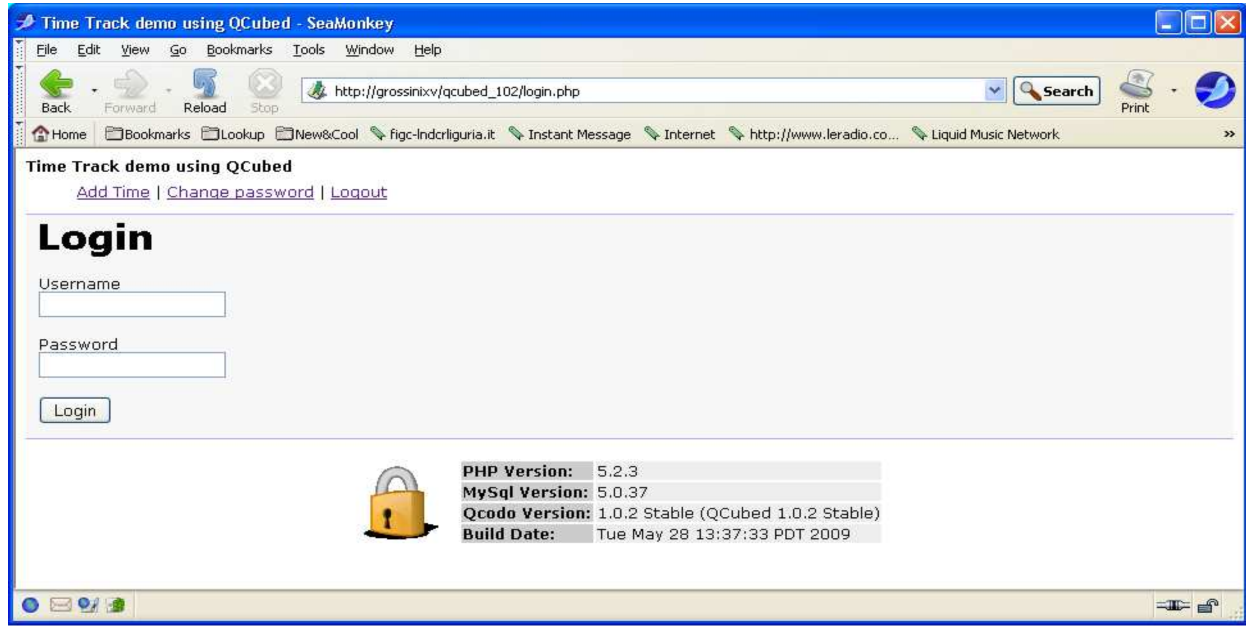
And finally



And the result will be:



ONE ARTIST'S TOUCH ON LOGIN SCREEN



Why not add some lines of code to have this login screen? Let we see how:

Add image for lock in the same dir ecory of our login.php

(name is lock_transparent.png)

Revised header.inc.php

```
<?php
  // Include prepend.inc to load QCubed
  require('includes/configuration/prepend.inc.php'); /* if you DO NOT have "includes/" in your
include_path */
  // require('prepend.inc.php');
?>

<html>
<head>
<title>Time Track demo using Qcubed v.2.1.0</title>
<style>
DIV.controlPanel {
  background:#F7F7F7;
  margin:10px 0px;
  padding:0px 10px;
  border:#AAF 1px solid;
  border-left:0px;
  border-right:0px;
}

TD, BODY { font: 12px <?php echo QFontFamily::Verdana; ?>; }
.title  {  font: 30px  <?php  echo  QFontFamily::Verdana;  ?>; font-weight: bold;  margin-left:-
2px;}
.title_action { font: 12px <?php echo QFontFamily::Verdana; ?>; font-weight: bold; margin-bottom: -
4px; }
.item_divider  {  line-height:  16px;  }
.heading { font: 16px <?php echo QFontFamily::Verdana; ?>; font-weight: bold; } </style>
</head>
<body>
<div class="title_action">Time Track demo using QCubed</div><ul>
<? include "menu.inc.php" ?></ul>
```

new Login.tpl.php

```
  <?include "header.inc.php" ?>
  <div class="controlPanel">
  <?php $this->RenderBegin() ?>
  <div class="title">Login</div>
  <br class="item_divider" />
  <?php $this->txtUsername->RenderWithName(); ?>
  <br class="item_divider" />
  <?php $this->txtPassword->RenderWithName(); ?>
  <br class="item_divider" />
  <?php  $this->btnLogin->Render()  ?>
  </div>
  <table align='center' cellpadding='5' >
<tr>
<td>
<img src="./lock_transparent.png">
</td>
<td>
<table>
<tr>
<td bgcolor='#CCCCCC'>
<strong>PHP Version:</strong></td>
<td bgcolor='#EEEEEE'>
<?php _p(phpversion()) ?></td>
</tr>
<tr>
<td bgcolor='#CCCCCC'><strong>MySql Version:</strong></td>
<td bgcolor='#EEEEEE'><?php _p(mysqli_get_client_info()); ?></td>
</tr>
<tr>
<td bgcolor='#CCCCCC'><strong>Qcodo Version:</strong></td>
<td bgcolor='#EEEEEE'><?php _p(QCUBED_VERSION); ?></td>
</tr>
<tr>
<td bgcolor='#CCCCCC'><strong>Build Date:</strong></td>
<td bgcolor='#EEEEEE'>Tue May 28 13:37:33 PDT 2009</td>
</tr>
</table>
<td>
</tr>
</table>
<?php $this->RenderEnd() ?>
<? include "footer.inc.php" ?>
```

Summary

Using examples pages as tutorial we went in depth in Qcubed events and classes.

We changed DB structure and this involved little modification on code thanks to design af application with directory related to tables, the Panel code created in dashboard and the magic index.php and index.tpl.php.

As exercise you can add logic to create and edit Panels for login table using what we did in chapt.07 for user table (*crypt password and add textbox for password2 to check password before crypting*).

CHAPTER 9: QCUBED OBJECTS REFERENCE GUIDE

See API Docs/

GENERAL PACKAGE

Class Hierarchy

INTERFACES

QTranslationBase

CLASSES

JavaScriptHelper
QApplicationBase
QArchive
QBaseClass
QBorderCollapse
QBorderStyle
QBrowserType
QCache
QCalendarType
QCallerException
QCallType
QCausesValidation
QCrossScripting
QCryptography
QCryptographyException
QCursor
QDateTime
QDateTimeNullException
QDateTimePickerFormat
QDateTimePickerType
QDateTimeSpan
QDirectoryToken
QDisplayStyle
QEmailAttachment
QEmailException
QEmailMessage
QEmailServer
QEmailStringAttachment
QErrorAttribute
QFile
QFileAssetType
QFileFormStateHandler
QFolder
QFormStateHandler
QGridLines
QHorizontalAlign
QI18n
QImageType
QIndexOutOfRangeException
QInstallationValidationResult
QInstallationValidator
QInvalidCastException
QInvalidFormStateException
QJsClosure
QLexer
QMetaControlArgumentType
QMetaControlCreateType
QMimeType
QOptimisticLockingException
QOverflow
QPartialQueryBuilder
QPoParserException
QPosition
QQ
QQAggregationClause
QQAssociationNode
QQAverage
QQBaseNode
QQClause
QQCondition
QQConditionAll
QQConditionAnd
QQConditionBetween
QQConditionComparison
QQConditionEqual
QQConditionGreaterOrEqual
QQConditionGreaterThan
QQConditionIn
QQConditionIsNotNull
QQConditionIsNull
QQConditionLessOrEqual
QQConditionLessThan
QQConditionLike
QQConditionLogical
QQConditionNone
QQConditionNot
QQConditionNotBetween
QQConditionNotEqual
QQConditionNotIn

QQConditionNotLike

QQConditionOr

QQCount

QQDistinct

QQExpand

QQExpandAsArray

QQExpandVirtualNode

QQGroupBy

QQLimitInfo

QQMaximum

QQMinimum

QQNamedValue

QQNode

QQOrderBy

QQReverseReferenceNode

QQSubQueryCountNode

QQSubQueryNode

QQSubQuerySqlNode

QQSum

QQuery

QQueryBuilder

QQueryExpansion

QQVirtualNode

QRegex

QRemoteAdminDeniedException

QRepeatDirection

QRequestMode

QResizeHandleDirection

QRssCategory

QRssFeed

QRssImage

QRssItem

QSelectionMode

QSessionFormStateHandler

QSoapMethod

QSoapParameter

QSoapService

QStack

QString

QTextAlign

QTextMode

QTimer

QTranslationPoParser

QType

QUndefinedPrimaryKeyException

QUndefinedPropertyException

QUpdateUtility

QVerticalAlign

FUNCTIONS

PrintInstructions

QcodoHandleError

QcodoHandleException

QDateTimeErrorHandler

QUpdateUtilityErrorHandler

QUpdateUtilityFileSystemErrorHandler

QUpdateUtilityFileSystemErrorHandlerForDelete

QUpdateUtilityFileSystemErrorHandlerForRename

stripslashes_array

_b

_f

_i

_p

_t

__database_check_error

__ob_callback

FILES

- cli_prepend.inc.php
- codegen.inc.php
- DisableMagicQuotes.inc.php
- error.inc.php
- JavaScriptHelper.class.php
- library.inc.php
- QAccordion.class.php
- QAccordionBase.class.php
- QApplicationBase.class.php
- QArchive.class.php
- QAutocomplete.class.php
- QAutocompleteBase.class.php
- QBaseClass.class.php
- QCache.class.php
- QCheckBoxColumn.class.php
- QCodeGen.class.php
- QControlProxy.class.php
- QConvertNotation.class.php
- QCryptography.class.php
- qcubed.inc.php
- qcubed_downloader.inc.php
- qcubed_orm_only.inc.php
- qcubed_updater.inc.php
- QDatepicker.class.php
- QDatepickerBase.class.php
- QDatepickerBox.class.php
- QDatepickerBoxBase.class.php
- QDateTime.class.php
- QDateTime.legacy.class.php
- QDateTimeSpan.class.php
- QDialog.class.php
- QDialogBase.class.php
- QDraggable.class.php
- QDraggableBase.class.php
- QDroppable.class.php
- QDroppableBase.class.php
- QEmailServer.class.php
- QExceptions.class.php
- QFile.class.php
- QFileFormStateHandler.class.php
- QFolder.class.php
- QFormStateHandler.class.php
- qform_validate_unique.tpl.php
- QI18n.class.php
- QI18nTests.php
- QInstallationValidator.class.php
- QJqButton.class.php
- QJqButtonBase.class.php
- QLexer.class.php
- QMimeType.class.php
- QPaginator.class.php
- qpanel_validate_unique.tpl.php
- QProgressbar.class.php
- QProgressbarBase.class.php
- QQuery.class.php
- QQueryExpansion.class.php
- QRegex.class.php
- QResizable.class.php
- QResizableBase.class.php
- QRssFeed.class.php
- QSelectable.class.php
- QSelectableBase.class.php
- QSessionFormStateHandler.class.php
- QSlider.class.php
- QSliderBase.class.php
- QSoapService.class.php
- QSortable.class.php
- QSortableBase.class.php
- QStack.class.php
- QString.class.php
- QTabs.class.php
- QTabsBase.class.php
- QTimer.class.php
- QTranslationBase.class.php
- QTranslationPoParser.class.php
- QType.class.php
- QUpdateUtility.class.php
- validate_unique.tpl.php
- _enumerations.inc.php