

QCubed 1.0 RC1

Quickstart

rev c

Copyright 2008 © – Kristof Meirlaen

Used by permission. No part of this publication can be reproduced without the express written consent of the Editor.

Table of Contents

Table of Contents

Table of Contents.....	2
Introduction.....	3
Welcome to QCubed!.....	3
The Code Generator.....	3
The ORM Layer.....	3
The GUI Layer.....	3
Intended audience.....	3
Chapter 1: Installation.....	4
Prerequisites.....	4
Webserver.....	4
PHP 5.....	4
Database.....	4
Installation.....	4
Step 1: Retrieving and unpacking QCubed.....	4
Step 2: Moving files to their proper location.....	4
Step 3: Setting permissions.....	5
Step 4: Setting the DOCROOT_SUBFOLDER and database parameters.....	5
Step 5: Disable Magic Quotes.....	6
Step 6: verify configuration.....	7
Chapter 2: Introduction to QForms.....	8
Definition.....	8
Our first QForms application.....	8
Structure of a QForms application.....	8
Example breakdown.....	9
Adding items.....	10
Adding events.....	11
Definition – Explained:.....	12
Summary.....	12
Chapter 3: Introduction to Code Generation.....	14
What QCubed does for you.....	14
Code Generation of Data Objects.....	14
HTML Form Drafts.....	14
What you still have to do.....	14
Example.....	14
Database layout.....	14
Code generation.....	15
Overview of the generated files.....	15
Viewing the result.....	16
Chapter 4: Understanding the generated code.....	19
Data Objects.....	19
QForm Drafts and Dashboard.....	20
Where to go from here?.....	21
QCubed Examples Site.....	21

Introduction

Welcome to QCubed!

QCubed is a PHP5 Framework. The goal of the framework is to provide a unique alternative to other frameworks available, by making different architectural decisions. The two major pieces of the QCubed framework are the ORM layer and the GUI layer. Each layer plays a separate and unique role, and is designed to be able to operate independently of the other.

The Code Generator

The first major piece is the Code Generator. Because of the nature of PHP, a major efficiency step is to make short and concise code. The less code that is included on each request, the better. The Code Generator is a sub-system of QCubed that generates PHP code based on templates and data derived from the project data schema. These PHP files are then written out into the project and made available to use. This drastically increases the efficiency of the system.

The ORM Layer

The Object Relational Modeling layer provides an API, after code generation, to manipulate the database of your project. It includes full support for relational tables, even when the underlying database does not.

The GUI Layer

The GUI layer is based on an object called a QForm. A QForm is a PHP class that facilitates a state-driven GUI layer over the stateless HTTP protocol. Advanced Web 2.0 features like Ajax based controls allow creation of really cool websites. The GUI layer can easily be extended with additional QControls.

Intended audience

This document is intended for developers wishing to understand how QCubed works and who wish to learn how to use QCubed for their own development.

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, knowing PHP 5 and it's object-oriented approach is a huge advantage.

Chapter 1: Installation

Prerequisites

Webserver

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 also need one of the following database platforms installed:

- MySQL (version 4 or 5)
- MSSQL
- Postgres 8.0

Make sure that you have at least one database installed on the platform, and you have a user and a password to connect to the database.

Installation

Step 1: Retrieving and unpacking QCubed

QCubed can be retrieved directly from the QCubed website (<http://www.qcu.be>);

The file that is retrieved is packed as a zip file. Unpack the zip file to a temporary directory using your favorite tool.

After unpacking, the following files and directories are created:

- `_README.txt`: the installation instructions for QCubed.
- `LICENSE.txt`: the license under which QCubed is released
- `_devtools_cli`: this directory contains command-line-based drivers for QCubed development tools
- `wwwroot`: this directory contains all the files that are required to run QCubed within your webserver, as well as some example files to get you going.

Step 2: 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.

To install QCubed, we need to copy the files from the `wwwroot` directory above to the root

directory of our webserver. As most of you will already have something in the default directory, create a new directory in your document root and call it “qcubed”. Afterwards, copy all files from the wwwroot directory from the QCubed archive to this new directory.

For example, if your documentroot is under /var/www/html, the files from wwwroot should go in a directory /var/www/html/qcubed, resulting in the following directory structure:

```
/var/www/html/qcubed/  .
                       ..
                       assets
                       drafts
                       examples
                       includes
                       index.php
                       sample.php
                       sample.tpl.php
                       _devtools
```

Step 3: Setting permissions

Because the QCubed 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/qcubed
```

Where /var/www/html/qcubed is the directory where you copied the wwwroot files to.

Step 4: Setting the DOCROOT_SUBFOLDER and database parameters

The last step is to configure QCubed. The main configuration file for QCubed is located in the includes directory of QCubed and is named 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/qcodo/wwwroot');
define ('__VIRTUAL_DIRECTORY__', '');
define ('__SUBDIRECTORY__', '/qcubed');
```

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__', '/var/www/html');
define ('__VIRTUAL_DIRECTORY__', '');
define ('__SUBDIRECTORY__', '/qcubed');
```

- If you are on windows, be sure to use forward slashes at all times. For example, if your document root is located in c:\inetpub\wwwroot, you could use c:/inetpub/wwwroot for the `__DOCROOT__`.

We also need to set the database connection properties. This is done in the same configuration.inc.php file.

Locate the following section, and set the appropriate parameters for Server, Database, Username and Password

```
define('DB_CONNECTION_1', serialize(array(
    'adapter' => 'MySql5',
    'server' => 'localhost',
    'port' => null,
    'database' => 'test',
    'username' => 'root',
    'password' => '',
    'profiling' => false)));
```

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

- MySql (MySQL v4.x, using the old mysql extension)
- Mysqli (MySQL v4.x, using the new mysqli extension)
- Mysqli5 (MySQL v5.x, using the new mysqli extension)
- SqlServer (Microsoft SQL Server)
- PostgreSQL (PostgreSQL 8)

Step 5: Disable Magic Quotes

QCubed works best with magic quotes disabled. If enabled, the QCubed start page would show an error message like “WARNING: magic_quotes_gpc and magic_quotes_runtime need to be disabled”.

Check your php.ini settings for the values of these settings. Set them to “off” if they are enabled and restart your webserver for the changes to take effect.

php.ini sample configuration:

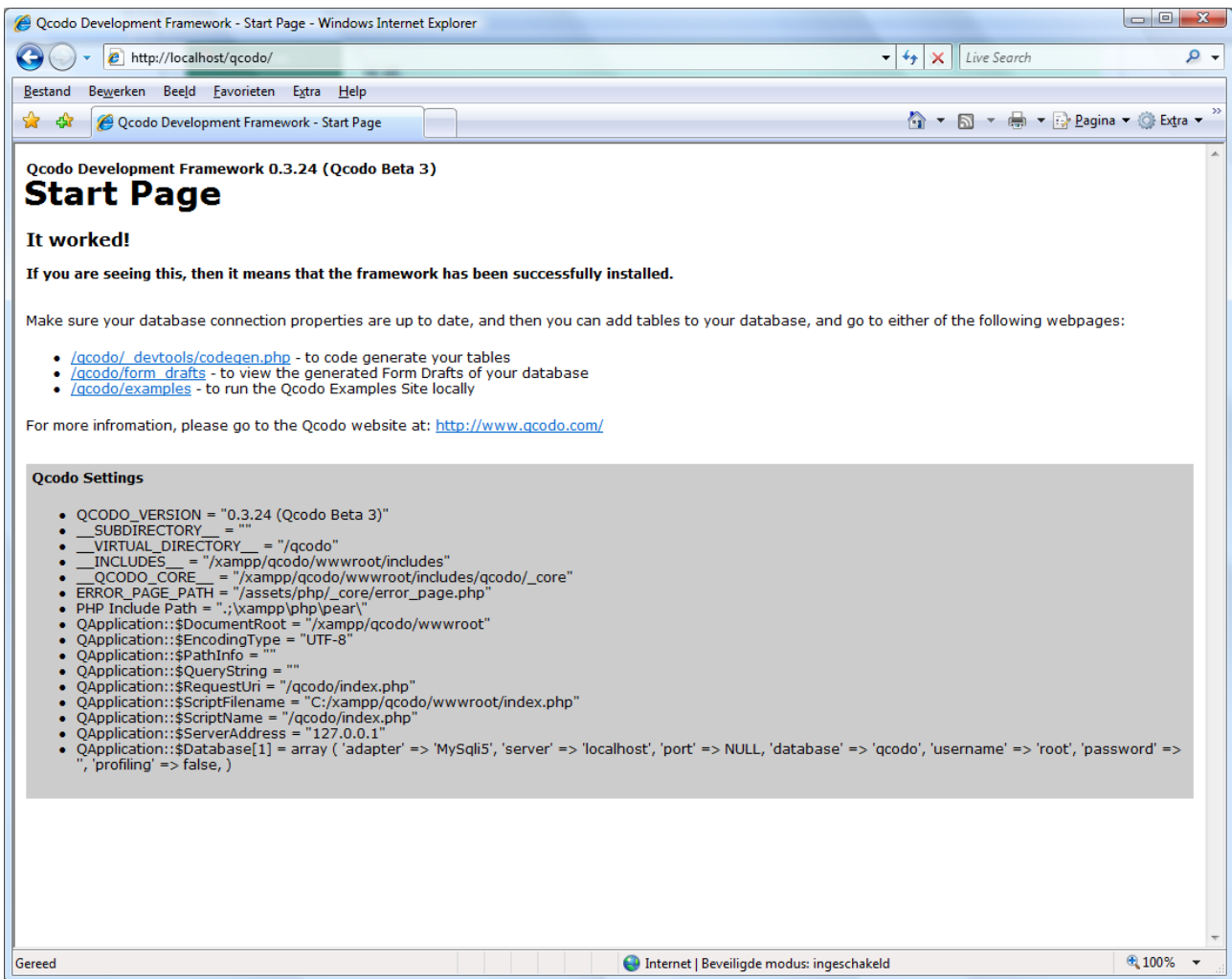
```
; Magic quotes
;

; Magic quotes for incoming GET/POST/Cookie data.
magic_quotes_gpc = Off

; Magic quotes for runtime-generated data, e.g. data from SQL, from exec(), etc.
magic_quotes_runtime = Off
```

Step 6: verify configuration:

We should now be able to go to our webserver and view the QCubed start page:



UPDATE REQUIRED

Chapter 2: Introduction to QForms

Definition

As much as everybody likes the code generation of QCubed, an equally important (if not more important) piece of QCubed is QForms. As the result of the Code generation piece of QCubed are QForms, we should have a basic understanding on what QForms is, and how QForms work before covering the code generation.

So what is QForms? From Mike Ho (creator of QCodo):

“QForms is an object-oriented, stateful, event-driven architecture for forms rendering and handling”

Nice... but what does it mean?

Our first QForms application

Let me try to explain what QForms is and how QForms works by creating a sample application. We must create a page with a button. The page should also display how many times the button has been clicked.

Structure of a QForms application

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

- example1.php
- example1.tpl.php

example1.php contains the **logic** of our application (also known as the controller in the MVC approach), and example1.tpl.php is the **presentation** layer of our application (also known as the 'view' in the MVC approach).

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

example1.php:

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

    class Example1 extends QForm {

        protected function Form_Create() {

        }

    }

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

example1.tpl.inc:

```
<html>
<head>
  <title>Our first QCubed application</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_once('includes/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 QForms.

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

We are going to create a QForm based application, so we create a new Object "Example1". We derive it from QForm.

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

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

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

This calls the parent (QForm) Run method, causing our application to execute.

This ends our example1.php page.

For example1.php.inc, you can see that the file contains mostly HTML. The only thing that is non-html are the following:

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

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

Basically, this will render a <form> html element in which all other QCubed components that are visible to the end user will be placed.

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.

Adding items

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 and 2 objects (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 initialisation of our form goes:

```
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:

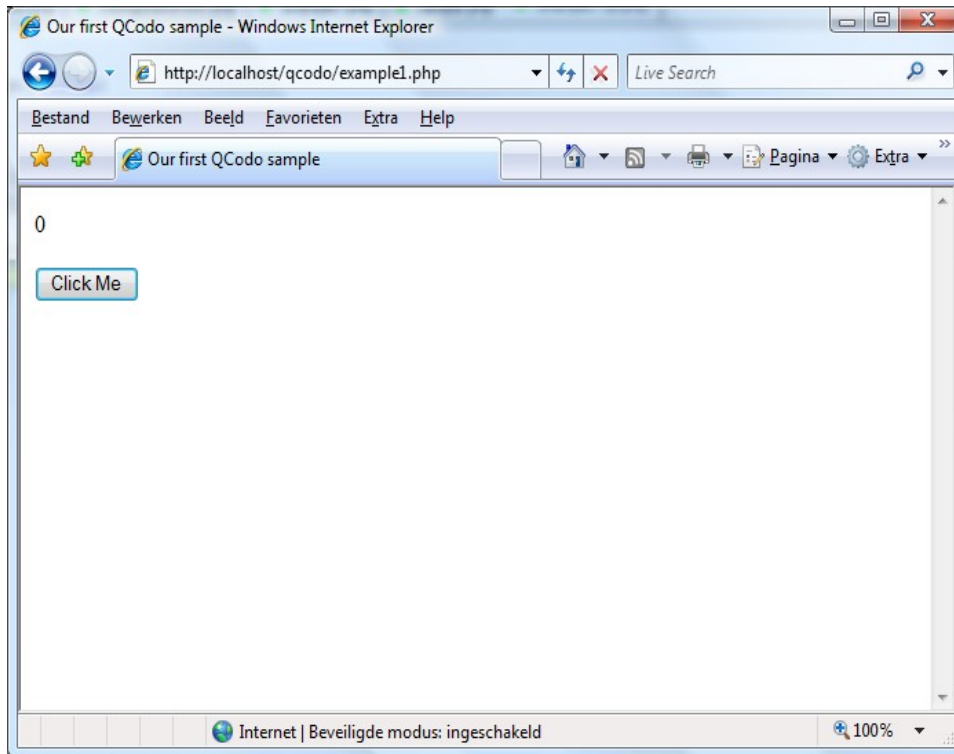
```
...
<?php $this->RenderBegin(); ?>

    <?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* something.

Adding events

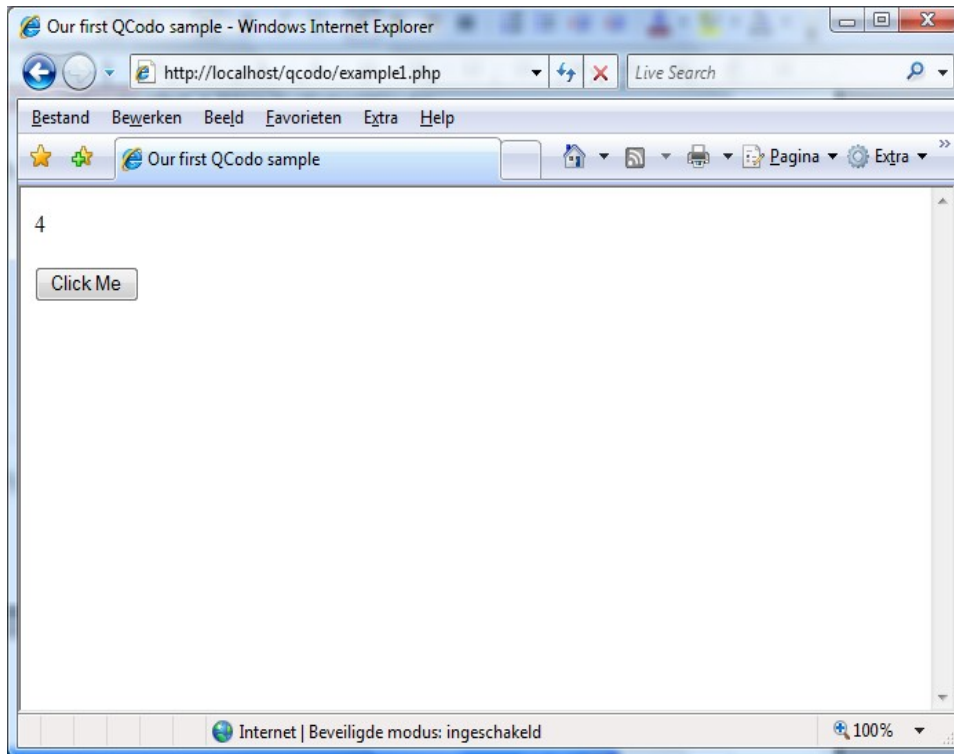
Now we have the “event-driven” part: when we click the button, we want the variable `$intNumberOfClicks` to be incremented, and the display to be updated. What we need to do is to assign an event to `$btnButton`. So in the `Form_Create()`, we add:

```
...  
    $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.



Definition – Explained:

So, let us take the definition from Mike:

“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.
- statefull: notice that QCubed “remembers” the value of the intNumberOfClicks. We did not have to store it in a session or database ourselves
- event-driven: assign actions to our objects (such as in our example a click-event), and execute some code based on these events.
- forms rendering and handling: notice that we did not have to code any <form> or <input> tags. QCubed handles this for us.

Summary

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 object 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 will generate

- Code Generation of Data Objects (the 'Model' in the MVC concept)
- the HTML Form Drafts, or basic scaffolding through the use of code generated QForms.

Code Generation of Data Objects

For each table in your database, QCubed will generate the code to translate the table to an object (class) that contains all the basic CRUD-type method 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.

This is all accomplished by analyzing the data structure of your table.

HTML Form Drafts

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

These generated pages are often a great starting point for your application.

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-driven 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.

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

```
CREATE TABLE `todolist` (  
  `id` int(11) NOT NULL auto_increment,  
  `description` varchar(100) NOT NULL,  
  PRIMARY KEY (`id`)  
) ENGINE=InnoDB DEFAULT CHARSET=latin1
```

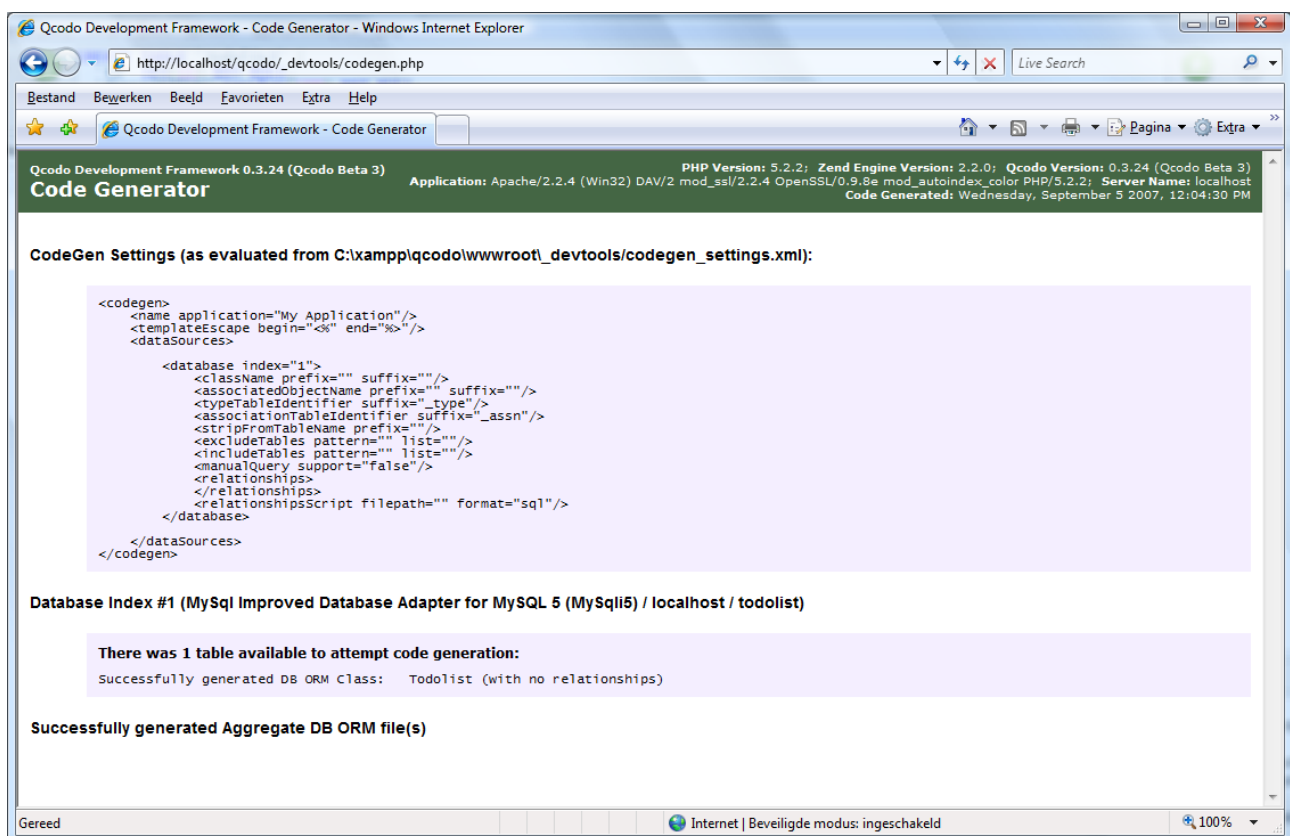
Code generation

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:

- [/qcubed/_devtools/codegen.php](#) - to code generate your tables
- [/qcubed/form_drafts/](#) - to view the generated Form Drafts of your database
- [/qcubed/examples/](#) - to run the QCubed Examples Site locally

To generate the code for, click the first link ['/qcubed/_devtools/codegen.php'](#).



If all goes well, you should see "No errors reported".

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 14 files:

- includes/data_classes/Todolist.class.php
- includes/data_classes/generated/TodolistGen.class.php
- drafts/todolist_edit.php
- drafts/todolist_list.php
- drafts/generated/todolist_edit.tpl.php
- drafts/generated/todolist_list.tpl.php
- drafts/dashboard/TodolistEditPanel.class.php
- drafts/dashboard/TodolistEditPanel.tpl.php
- drafts/dashboard/TodolistListPanel.class.php
- drafts/dashboard/TodolistListPanel.tpl.php
- data_meta_controls/TodolistDataGrid.class.php
- data_meta_controls/TodolistMetaControl.class.php
- data_meta_controls/generated/TodolistDataGridGen.class.php
- data_meta_controls/generated/TodolistMetaControlGen.class.php

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

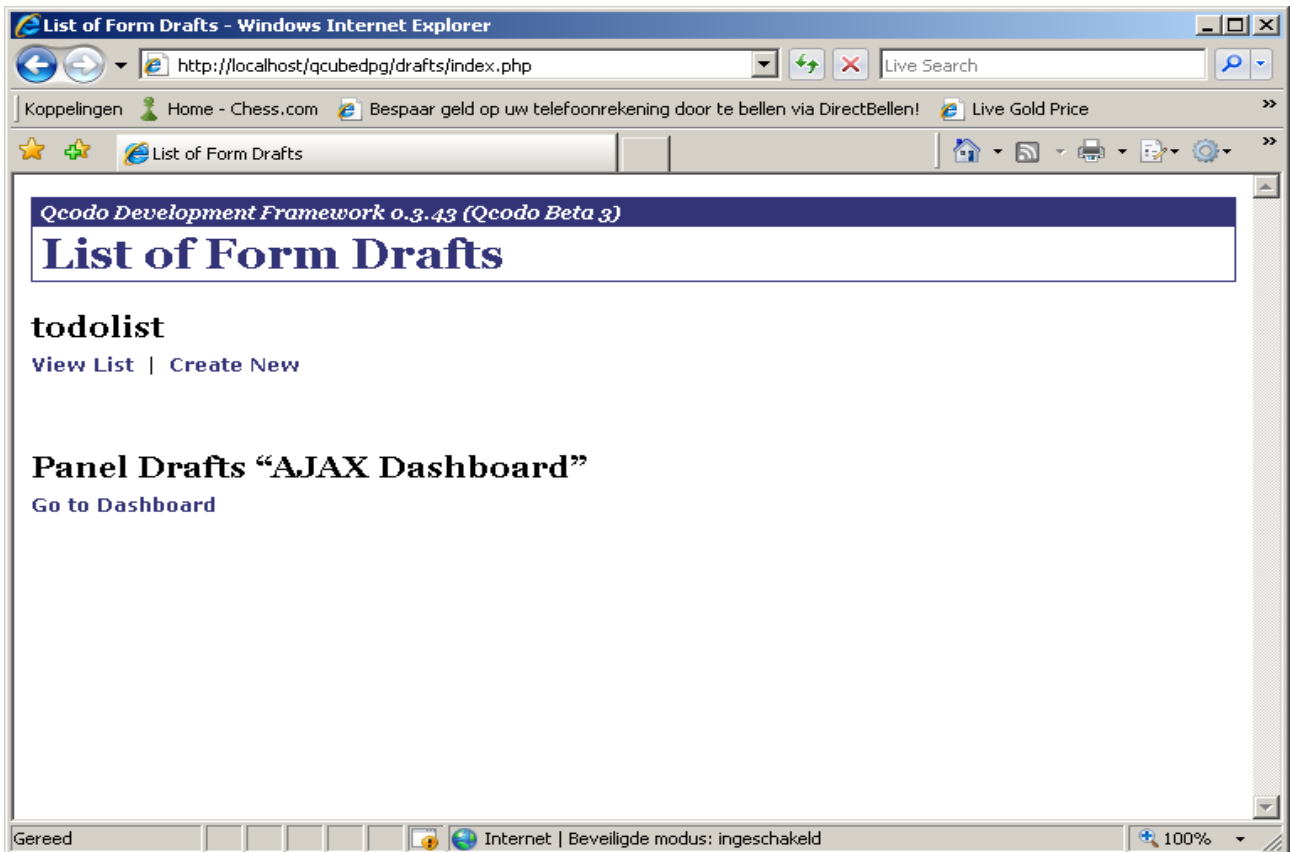
1. Code Generation of Data Objects
2. the HTML Form Drafts
3. Meta Controls

The first 2 files in the list are the result of the Code Generation of the Data Object (Model) for our Todo list. The files under /drafts are code generated QForms to allow us to do the basic CRUD functionalities on the objects. The last 4 files under data_meta_controls are code generated meta controls. The draft files use a MetaControl which, itself, contains the links between a set of QControls and its underlying data object.

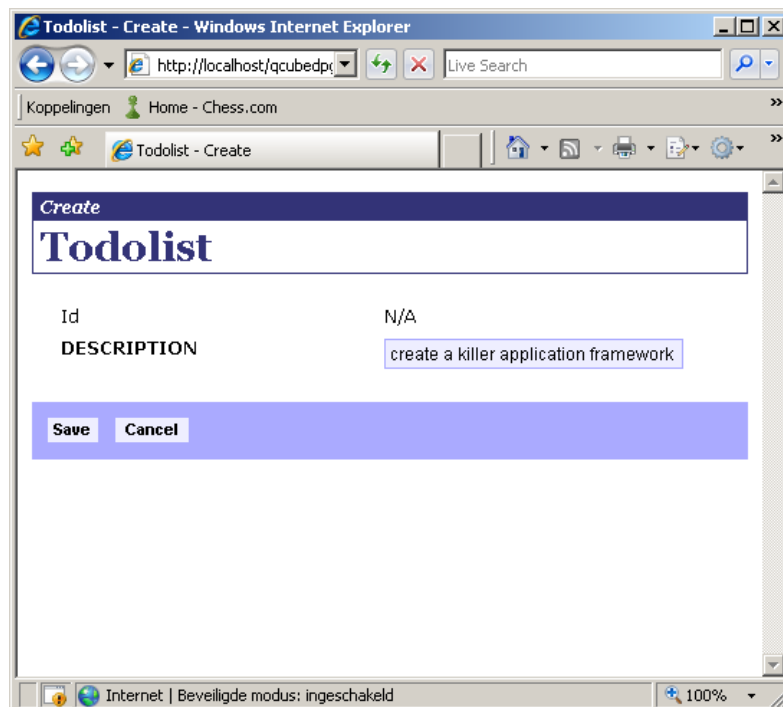
Viewing the result

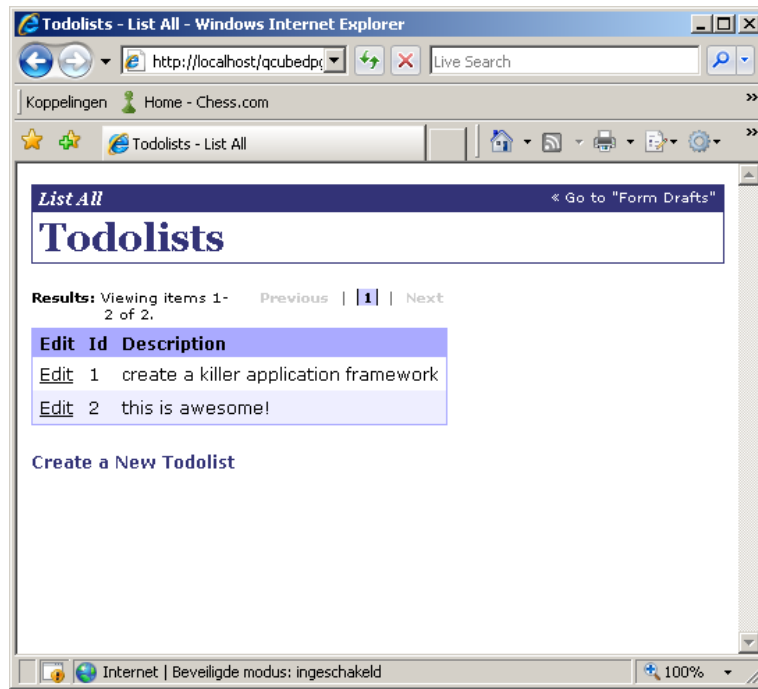
To see the result of the code generation, go back to the homepage of your QCubed installation and now, click the second link : /qcododrafts/

It will show the following page:



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 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, and pagination is also included.

Chapter 4: Understanding the generated code

In chapter 3, we have learned how to generate code from an existing database model using QCubed. In this chapter, we 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/data_objects`) and `TodolistGen.class.php` (in `includes/data_objects_generated`).

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

- `Load($intId)`
- `LoadAll($objOptionalClauses = null)`
- `CountAll()`
- `Save($bInForceInsert = false, $bInForceUpdate = false)`
- `Delete()`
- `DeleteAll()`
- `Truncate()`

The `TodolistGen.inc` also contains code to allow access to 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 `Todolist` 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 in `Todolist.inc`, 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 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 frontend
- an advanced ajax dashboard

To display the drafts pages, 4 files are created: 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 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).

The dashboard is an advanced code generated, ajax powered way of providing CRUD functionality. Instead of using separated QForms, this application uses QPanels, which have 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 of the above lines, you would type (mctTodolist is the Todolist Meta control object),

```
$txtName = $mctHobby->txtName_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 data_meta_control folder. It is your choice what you use and what you do not. You are not tied to certain features with this code generation.

Where to go from here?

QCubed Examples Site

This is a collection of many small examples that demonstrate the functionality in QCubed. Later examples tend to build upon functionality or concepts that are discussed in prior ones, which allows the Examples site to be viewed as a quasi-tutorial. However, you should still feel free to check out any of the examples as you wish.

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

You can find the examples on your local installation of QCubed:

<http://yourhost/qcubed/examples>