Qore

Administration and Developer Guide

Version 1

Table of Contents

Overview

Qore is a simple MVC implementation to build sites on. The implementation takes a minimalist approach

in that the only thing it provides is the bare minimum that is needed. Currently, it provides:

1. Router/Dispatcher: to route requests to "controllers"
2. Templating engine (Twig) to render the views.
3. "Packs" to keep all code modular and re-usable
4. Session Manager
    1. file based sessions - stored in {Qore Install Dir}/tmp/session directory
    2. DB based sessions - see {Qore Install Dir}/config/db/sessions-mysql.sql for the needed table
5. supports session flash messages for both informational and errors
6. Dependency Injection (Pimple) for testability and re-useability
7. Factory's to auto select the correct database driver & table adapter to access databases
8. Plug-able/extendable Authorization and Authentication layer (TODO)

The framework uses and relies upon PHP namespaces - so PHP 5.3+ is a must.

The framework resides in libraries\Qore.

Request Flow

1. all incoming requests go to index.php in the webroot (which is the only directory that is web accessible) which then calls config/bootstrap.php and config/routes.php
2. config\routes.php instantiates he router (libraries\Qore\router.php) which takes care of cleaning URLs, and routing requests to a specific controller.
3. the appropriate controller is called based on the registered controller search path ($router->registerControllerPath()).
4. the controller does it's thing (loads models to get data from the DB, processes it etc..)
5. the controller hands data to a Twig view ($this->twig->render()).
6. the Twig view displays the data to the user

Products Used

- PHP: (http://www.php.net)
- MySQL: (http://www.mysql.com)
- Twig PHP Templating Library: (http://twig.sensiolabs.org/doc/templates.html)
- Pimple Dependency Injection: (http://pimple.sensiolabs.org/)

Installation

Overview

- copy all the file to a directory somewhere.
- configure Apache to serve up the webroot directory.
- edit config\config.php
- edit config\routes.php (setups up the router)

Requirements

- Apache
  - mod_php
  - mod_rewrite
- PHP
  - 5.3
- Database (Currently MySQL only)

Apache Requirements

- Apache 2.2 with:
  - mod_rewrite
  - mod_php

Apache Vhost Configuration Example:

Please note that “AllowOverride All” and “Options FollowSymlinks” are needed in your apache config:

```
<VirtualHost *:80>
    ServerAdmin webmaster@domain.com
    DocumentRoot "/var/www/BzStatsWeb/webroot"
    ServerName mydomainname

    ErrorLog "logs/BzStatsWeb-error.log"
    CustomLog "logs/BzStatsWeb-access.log" common

    <Directory "/var/www/BzStatsWeb/webroot">
        AllowOverride All
        Options FollowSymLinks
        Order allow,deny
        Allow from all
    </Directory>
</VirtualHost>
```

The included .htaccess file in the webroot gives friendly URL's, and routes everything properly to index.php.

Apache Alias example:

If you would rather 'alias' webroot from an existing site (meaning it will not be served from the site's root)

then you must set it up as follows:

```
alias /myAlias "/path/to/project/webroot"
<Location /myAlias>
    Options FollowSymLinks
    Order allow,deny
    Allow from all

    <IfModule mod_rewrite.c>
        RewriteEngine On

        RewriteCond %{REQUEST_FILENAME} !-f
        RewriteCond %{REQUEST_FILENAME} !-d

        RewriteRule ^(.*)$ index.php?url=$1 [PT,L]
    </IfModule>
</Location>

also, in webroot/index.php, make sure you set the SITE_SUBDIR constant to the
alias you defined above:

...
define('SITE_SUBDIR', '/myAlias');
...
```

Apache No Mod-Rewrite Setup

if you don't setup the mod_rewrite stuff, you can make things work URLs like the following:

```
if webroot is the root of the site:
http://mysite.com/index.php/<controller>/<method>/arg1/arg2/.../argN/

if webroot is an alias:
http://mysite.com/myAlias/index.php/<controller>/<method>/arg1/arg2/.../argN/
```

PHP Requirements

Required PHP extensions:

- php_pdo_mysql
- php_mysqli

Development

Packs

Development with Qore starts at the “Pack” level. Packs are really just a way to organize and re-use code, so it's where all your code should be organized and placed.

Packs:

- all development should be placed in a "Pack”.
  - A "Pack" is simply a directory under "packs"
  - copy the "skel" directory to get the recommended directory structure
  - each pack should be registered in config/config.php:

```
/**
 * Registered Packs
 */
$cfg['packs'] = array('packone', 'packtwo', ...);
```

- packs contain:
  - models
  - dbadapters

    - controllers
    - views
    - supporting PHP files (interfaces, abstract classes etc..)
- You should have a default pack (registered first in the $**cfg['packs']** array)
    - the default pack is searched first when looking for controllers
- if you plan on distributing your pack, make sure that the controller URL's are unique
    - it's a good idea to prefix your controller class names and php files with your pack name in this case
- all folders/filenames should be lowercase
- controller class name must be the same name as the file (without the PHP) + Controller (camel casing)
    - example: news.php has a NewsController class
        - class NewsController extends \Qore\Controller {}
    - all controllers that need a view, should extend the Controller class (\Qore\Controller)
    - all controllers which do not need a view (maybe an API type of controller) should extend BaseController (Qore\BaseController)
        - BaseController does not instantiate Twig - so no view/twig objects exist.
        - You can use this to control exactly what to output (no view templates will be used)
- all publicly accessible methods must end in _public.
    - for example: news.php with class NewsController has a display_public($args){} method.
    - only methods that end in “_public” are accessable publicly through the router.
        - libraries/Qore/router.php which config/routes.php instantiates
    - so a URL of `http://mydomain.com/news/display/10` would call the news controller
        - <controller_path>/news.php would need to have a “display_public(array $args){}” method
        - “10” would be passed as an argument to the display() method (“$args[0] = 10” in this case)

Along with \Qore\Router enforcing that all public accessible method's must end with “_public”, by default, the controllers themselves will not allow any public method to

execute, as “\Qore\Router” check's the controllers execution state ($controller->getExecutionState()).

So, in order for public methods to be allowed to execute, the developer must satisfy two criteria:

1. all public methods must end in “_public”
2. all public methods must have their execution state to true ($this->setExecutionState())

You can set the execution state using one (or both) of the following to allow/deny execution to all/any controller method:

1. make all “_public” methods in a given controller executable by overriding the controller's “__pre()” method:

```
public function __pre() {
    parent::__pre();
    $this->setExecutionState(true);
}
```

2. mark a specific “_public” method as executable:

If your method is “display_public()”, then you can define “display_public_pre()”, and set “$this->setExecutionState” to true there:

```
public function display_public(array $args) {
    //regular business logic/controller code;
}

public function display_public_pre() {
    //marks display_public as executable,
    //   as this specific *_pre() method is called just before
    //   the corresponding *_public method is called
    //     in this case: it is called right before display_public()
    $this->setExecutionState(true);
}
```

This allows for much flexibility for anything the developer desires (authentication, authorization, or whatever).

Similarily, there is also “__post()” method (which, if it exists, is called right after all methods in the controlle), and a “*_public_post()” which is called after the corresponding method.

```
Class MyController extends \Qore\Controller {
   __pre() {
      //this is called before every <methodName>_public method is executed
   }
   __post() {
      //this is called after every <methodName>_public method is executed
   }

   default_public(array $args) {
      //this method could be publicly executable (as long as it's permitted)
   }

   default_public_pre(array $args) {
      //this is called right before default_public is called
      //you could allow it to be exeutable by setting:
      $this->setExecutionState(true);

      //you can also do the reverse, or do whatever else meets your needs
      //you may have also noticed that the <methodName>_public_pre() is also
      //passed the same $args as it's corresponding <methodName>_public method.
      //so you could check that the required $args array elements exist
      //or do some validation to see if execution is permitted.
   }

   default_public_post() {
      //this is called right after default_public() is called.
      //you can do some cleanup stuff in here – or whatever suits your needs
   }

   default() {
      //this is just a normal method,
      //and can never ever be called by the router (called publicly)
   }

   doSomethingAllMyMethodsNeed() {
      //you can have supporting methods (as many as you like)
      //and since they don't end in _public, you can rest assured
      //that they can't ever be called publicly.
   }
}
```

The order of execution is:

1. __pre();
2. <methodName>_public_pre();
3. <methodName>_public(); (if executionState is true)
4. <methodName>_public_post();
5. __post();

Please note that the “<methodName>_public_pre()” method is passed the same arguments that will be passed to the “<methodName>_public()” method.

This may be useful in determining if if the request should be allowed or not..

Template Paths and Error Pages

Qore uses Twig for rendering it's views, and also provides some default templates for error pages.

In order for Twig to find the correct views, it will search through all the registered 'Packs' first (in the order they are defined), and then it will search the default view folder, which is the view folder in the root folder (where you installed Qore).

So, if you have the following Pack's registered in your config/config.php:

```
$cfg['packs'] = array('mypack', 'otherpack');
```

the following “view” search path will be registered:

1. <qore-root>/packs/mypack/views
2. <qore-root>/packs/otherpack/views
3. <qore-root>/views

The template that is found first, is used to render the page. This can be useful, as it allows you to override views should you need to, but it's also something you should be careful of.

By default, Qore only has a single error template, and a site-default template. You can override them in your default Pack (the first one listed) should you desire. Just remember that you will have to create the needed subfolders in order for them to be found. For example, to override the default error page, (which is used for various HTTP errors (404 (page not found), 500 (internal server error), etc..), which is located in <qore-root>/views/qore/error/error.html.twig, you must place it in a 'qore/error' directory under your Packs' views directory. Using the above example, you could place it in two places:

1. <qore-root>/packs/mypack/views/qore/error/error.html.twig
2. <qore-root>/packs/otherpack/views/qore/error/error.html.twig

both places would work. Although #1 would always match first – so you might as well place it there.

You can leave the default.html.twig in the default views folder, but it may be overwritten upon updates, so I would recommend copying it to the first registered Packs folder anyways (since that will also speed things up a tiny bit – less directories to parse through to locate it).

Error Handling

Qore comes with a custem Exception class (\Qore\Qexception), which enforces that you set an HTTP status/error code when you throw errors. This is nice, as it allows you to

define error messages, and at the same time control what type of HTTP error code the browser will receive. The available error codes that you can use are:

- 200 (status OK, not an error, but for minor errors, it may be desirable to use this)
- 400 (bad request)
- 401 (Unauthorized)
- 403 (Forbidden)
- 404 (Not Found)
- 500 (Internal Server Error)
- 501 (Not Implemented)

This should allow you a decent amount of flexibility when defining errors.

By default, Qore will use a single error page (<qore-root>/views/qore/error/error.html – unless you override it), to display all errors through PHP's try/catch error handling, and Qore will automatically set the proper HTTP error code that the developer selected.

This way 404's (for things that aren't found (no controller found, no method found etc..) are properly registered, and things like web-spiders (for search engines) can be happy, as they will get proper HTTP error/status codes when crawling your site.

To use this functionality, you simply have to use:

```
throw new \Qore\Qexception("Error Message", \Qore\Qexception::$BadRequest);
```

Please note that \Qore\Qexception has a bunch of static properties/fields that should be used to set the desired HTTP error codes. They should be self explanatory.

For third party libraries that simply throw a generic PHP exception, 500 (internal server error) is used.

Sessions

Qore comes with a complete session management layer.

Qore's full session management layer is a complete solution that ensures no duplicate session names will ever be created, for both file based sessions or database based sessions. For clustered/load-balanced webservers database sessions are recommended.

You can use file based sessions if you want, as they are quick and easy. (no extra setup is required).

If you want to use database sessions, then you will need the table defined in “{Qore Install Root}/config/db/sessions-mysql.ql”.

This can exist in any database you want, just make sure that in your `{Qore Install Root}/config/config.php`, that you configure your database instance, and you set `$cfg['sessions']['dbinstance']` to the name of your configured DB instance.

In your controller, you have full access to the sessions object through `$this->sessions`.

Qore's session management uses namespace-ish indexes to organize the PHP sessions array. It is heavily recommended that you use the session object to read/write to the session rather then directly through PHP's $_SESSION array. this way you will ensure that you don't overwrite/kill something that is critical (as Qore sets some session variables in order to work properly, and do basic security checks etc...). Qore's session object methods enforce namespace use to almost guarantee that your session variables will not conflict with another's. Just make sure you use a namespace that is more or less unique to you (call it the same as your Pack for example).

Also note that Qore tries hard to make sure that every session is valid for every request - specifically:

- the timestamp is new enough (not expired, or exceeding the defined lifetime of a session)
- that the fingerprint matches the original fingerpring at session create time (to try and ensure that the same browser is using this session).
- all the internal security checks are OK (Qore's required session variables exist etc..)

This way, by the time a controller is called, you can be fairly certain that at least at a basic level, the session is valid.

Qore will automatically kill, redirect (if configured), and create a new session if it detects an invalid session - before the router is even called.

Your controllers $session object ($this->sessions), has the following public methods that you can/should use:

- $this->session->set(namespace, key, val);
  - sets a session variable called “key” in the specified “namespace” with a value of “val”.
- $this->session->get(namespace, key);
  - retrieves the value for session variable `key` in `namespace`
- $this->session->getId();
  - returns the session_id of the current session
- $this->session->unsetkey(namespace, key);
  - unsets (deletes) the session variable `key` in `namespace`
- $this->session->unsetNamespace(namespace);

- deletes the entire namespace from the current session

- $this->session->varIsSet(namespace, key);
  - returns true is session variable `key` exists under `namespace`; otherwise false if it doesn't exist
- $this->session->kill();
  - completely kills the server side, and client side components of a session
  - this may also redirect the user to the configured redirect page in the sessions configuration in `config/config.php`
  - you can pass an array to control how you want kill to perform:
    - array('redirect' => true|false, 'redirectURL' => 'url/to/redirect/to', 'sendDelCookie' => true|false)
    - sendDelCookie is if we should send a delete cookie to the client for it's session_id
      - if you are redirecting - send the DelCookie
      - if you are not redirecting, then you don't need to send the DelCookie, as a new blank session will be auto-started
      - note that by default, `kill()` will figure out if it needs to send the DelCookie or not (depending on redirect or not) but you can use this to manually override and tell `kill()` to send it or not.
- $this->session->regenerate();
  - this will regenerate the `session_id` for the current session. it will delete the old session, but will move over the current sessions variables to the newly created session_id.
  - this should be called after something important happens in your application (example, logon, logoff, etc..).
  - Qore will automatically call this every N requests for security reasons. you can control when with $cfg['sessions']['regenSessionId']` in your `config/config.php`. This should help overcome session hijacking etc..
  - Please see the “known issues” section – as there is a known issue with the auto-regeneration feature.
- $this->session->setError(msg); $this->session->setFlash(msg);
  - sets the sessions error or flash messages that will be displayed the next time getError or getFlash is called (see below)
- $this->session->getError(); $this->session->getFlash();
  - retrieves the error or flash message that was set in the session
- this will also delete the error/flash messages from the session - so they can only be displayed once.

The session object also contains a fairly detailed/complex random string generator that you may want to use for other things, and is available to you:

- $this->session->genId();
- $this->session->genId(array());

The random string can have 'sections'. By default there are 4 sections.

Each section is separated by a hyphen by default.

Each section should not be longer than 275 characters long. But since you can have as many sections as you want (of different length) this should not be an issue, and you can use this to generate strings of thousands (millions?) of characters long if you really want to.

The default settings produce the same random strings that are used internally for session_id`s which are 70 characters long:

```
731M1c47d95768ucA280o3p9Ve0e51-b6ff28e5d37e-45ew3-fdW704Fc085027G768cK
```

if you want different lengths of sections (or maybe just 1 section), then

you can pass an array and specify the 'sections' key:

- array('sections'=>array(10))
  - produces a single section (single string) 10 characters in length

you can also pass a 'separator' key to tell genId() which character(s) to use to separate each section:

- array('separator'=>'####')
  - would use the string '####' to separate each random section.

Examples:

```
$this->session->genId(
   array('separator' => '_',
         'sections' => array(10, 20, 30, 40, 50, 60, 70, 80)));
```

```
```

this would produce a random string with 8 sections for a total of 367 characters (360 characters plus 7 underscore separators)

```
$this->session->genId(
   array('separator' => '',
         'sections' => array(275, 275, 275, 275, 275)));
```

This would produce what appears to be a single random string of 1375 characters (275 * 5 sections) with no seperator character) so each section is simply concatenated to the rest

```
$this->session->genId(array('sections' => array(50)));`
```

This would produce a single random string section of 50 characters in length. Since it's just a single section that was specified, the separator is not used so you don't need to set it to blank or anything. So as long as you want a single random character string that is less than 275 characters, you can simply use something like this.

Query Strings & Posts

The router takes care of doing a basic clean on any strings that are passed via the URL (GET), or through an HTTP post.

To see what is cleaned by default, please check `$cfg['allowedUrlChars']` in `config/config.php`.

Only the characters specified in `$cfg['allowedUrlChars']` are allowed in a GET/POST request.

Since the URL is chopped up by forward slashes and assigned to an array, and any POSTS are also assigned to an array, each array element that has any non-allowed characters are simply removed...completely. So if you see some array elements disappear, check to make sure that the characters entered are valid/allowed.

Each controller has:

- $httpQuery
- $httpQueryData
- $httpPost
- $httpPostData

If any http request has query(ies)/post data, then they are automatically moved to the controllers “$httpQueryData” (for URL query strings), or “$httpPostData” (for POSTs) arrays. These are available to the controller at all time ($this->httpQueryData, or $this->httpPostData).

Also, the controllers $httpQuery will be “true” if a request query is encountered, and $httpPost will be “true” if post data is detected.

So you can easily use these to detect if there is any data that came in through Post of Get URL queries.

(and if you use this in combination with the controllers `<method>_public_pre()` methods, you can easily enforce a type of access to a method depending on GET/POST data..)

example: make a controller method only allow POSTed data:

```
    public function main_public_pre() {
        if (!this->httpPost) {
            $this->setExecutionState(false);
            throw new Exception("Sorry, You can only Post data here");
        }
    }
```

POSTs are easy, since PHP already has them in a $_POST array, the array is simply cleaned (at a basic level), and given to the controller.

For POSTs, the indexes match the form element names that were used to post the data.

For URL query string's, it's a little more complex, and I think the best way to understand it, is through a good example:

```
URL: http://my.domain.com/controller/method/arg1/arg2/?
name=ian&pass=123&options=test&options=test2&options=test3/arg3?test=6/arg4


\Qore\Router will parse this into the following $httpQueryData array:
    array
        'name' =>
            array
            0 => string 'ian' (length=3)
        'pass' =>
            array
            0 => string '123' (length=3)
        'options' =>
            array
            0 => string 'test' (length=4)
```

```
            1 => string 'test2' (length=5)
            2 => string 'test3' (length=5)
        'arg3' =>
            array
            'test' =>
                array
                0 => string '6' (length=1)

\Qore\Router will also hand the following data to the controller's method as
it's $args:
        array
            0 => string 'arg1' (length=4)
            1 => string 'arg2' (length=4)
            2 => string 'arg4' (length=4)

Accessing data from the above example url in a controller:

   public function <methodName>_public(array $args) {
      echo $args[0]; // 'arg1'
      echo $args[1]; // 'arg2'
      echo $args[2]; // 'arg4'

      echo $this->httpQueryData['name'];        //ian
      echo $this->httpQueryData['options'][1];  //test2
   }
```

Things to note:

- each and any url section that does not have a query string in it, is unaffected, and passed to the method as it's $args
- each url section that **does** have a query string detected, is removed from the $arg array, and moved to the controllers $httpQueryData array.
  - the controllers $httpQuery is set to true in this case
- also note arg3 in the URL. since the arg3 url section is: “arg3?test=6”, the index in the `$httpQueryData` array is named “arg3”, and a child array is assigned to it which contains the actual query parameters.
- also note the “options” index. Since it has many values in the query string, they are all given to the same index in the “$httpQueryData” array.
  - this is good for checkboxes etc..
- note that arg4 in the URL is passed to the controller method's $args.
  - This is important as it enforces the fact that each url section that contains query dat is removed from the $args array, and any non-query data sections are kept and the array renumbered in order for it to appear nicely (no gaps will be in the array as for as indexes go).

so, you can either put a query string directly after a forward slash:

```
http://my.domain.com/controller/method/?query=string
```

or assign an index for a particular query string:

```
http://my.domain.com/controller/method/httpQueryDataIndexKey?query=string&key2=val2
```

Dependency Injection & Factories & Database Access

Please note that Qore's database access makes heavy use of the built-in Dependency injection and Factories, so they will be used as examples to introduce how D.I. and Qore's Factories work together.

That said, the D.I container can be used for any other purposes that make sense, in order to obtain loosely coupled objects

Dependency Injection/Inversion Of Control

Dependency Injection is handled by the extremely simple `Pimple` class:

- libraries/Qore/ioc.php instantiates and wraps the Pimple DI container (it is called by config/bootstrap.php)
- config/config.php will look for an ioc.php file in the root of each registered “Pack”.
  - you use `ioc.php` in the root of your Pack to Register your DI objects
- You can use `IoC::Register()`, and `IoC::Get()` to set and retrieve your DI objects
- You can configure it so that objects are shared (re-used) or not
  - if shared, then the same object is returned each time a caller asks for it
  - if not, then a new object is created each time a caller asks for it.

You register your objects in the DI container by passing simple anonymous (lambda) functions to IoC::Register()

The Syntax is: `IoC::Register('keyName', lambda, bool $share)`

Note: bool $share determines if this object will be shared or not (true = shared, false = not shared).

```
IoC::Register(
    'packName.db.target.stats',
    function ($c) {
```

```
        $db = $GLOBALS['cfg']['packName']['dbInstance'];
        return \Qore\Factory\DbFactory::build(
            'PackName',
            $db,
            'stats',
            $c["db.$db.connection"]);
    },
    true
)};

IoC::Register(
    'packName.statsmodel',
    function ($c) {
        return
\Packs\packName\Models\StatsModel($c['packName.db.target.stats']);
    },
    true
);
```

The above examples also show that you can access the DI container within the anonymous function, and that all the registered objects will be shared.

`$c` is the Pimple container instance in the above examples. This way you can handle dependencies by creating a chain.

with the above examples, if a caller issues `IoC::Get('packName.statsmodel');`:

- the DI container will first satisfy all dependency objects:
  - create or re-use the `packName.db.connection` object
  - pass the `packName.db.connection` to the `packName.db.target.stats` object and create/re-use it
- pass the `packName.db.target.stats` to the 'packName.statsmodel' object, and return that object to the caller.

So, with a simple IoC::Get() call, we satisfy all the dependencies for all our objects with the help of Qore's factories.

Factories

Qore includes two factories that the Dependency Injection layer (\Qore\ioc.php) should take advantage of:

- \Qore\Factory\DbConFactory
- \Qore\Factory\DbFactory

DbConFactory

```
\Qore\Factory\DbConFactory::build($dbinstance, $default = true, $pack = '');
```

This factory is responsible for creating DB connection objects.

For example, if you ask for `\Qore\Factory\DbConFactory::build('default');`:

- DbConFactory will search the `$cfg['db']['default']['type']` array key
- if that array key is set to `mysql`:
  - Qore\Dbcon\DbMysql will be instantiated
  - Qore\Dbcon\DbMysql is in libraries/Qore/dbcon/dbmysql.php
- it also loads the $cfg['db']['default'] array to obtain the connection details

since by default, \Qore\Facory\DbConFactory is configured to load the default DB connection object it will always just instantiate the appropriate default object (which should be good enough for most needs).

if for whatever reason they don't meet your needs, you can use it to create custom objects for your pack:

```
\Qore\Factory\DbConFactory::build($dbinstance, false, $pack = 'MyPackName');
```

if `$cfg['db'][$dbinstance]['type']` is set to `'mysql'`:

- packs\mypackname\dbcon\dbmysql.php` is included
  - the class name should be `DbMysql`
- the `$dbinstance` will be passed to DbMysql() constructor
  - DbMysql should implement `\Qore\Unreal\iDb` (`libraries\Qore\unreal\idb.php`)
  - DbMysql should check the $cfg['db'][$dbinstance] array for all the connection details.
- the constructor should try and connect to the database, and set a handler

you can simply copy `libraries/Qore/dbcon/dbmysql.php` and customize it to your needs for another database type, it should be renamed appropriately.

DbFactory

the DbFactory is responsible for returning a specific dbAdapter for a pack.

the DbFactory is passed 4 parameters: $pack, $dbInstance, $target, $pdoDbHandler.

- $pack is the pack that the dbadapter belongs to.
- $dbInstance is the database instance set in config.php that you want to use.
  - ($cfg['db'][<instance>]['type'] etc..)
- $target is a specific string that differentiates this dbadapter.
  - this way you can separate your DB calls into multiple dbadapters.
  - each dbadapter can be specific for certain types of DB queries (separate by tables, or perhaps types of queries, or data domains etc..)
  - DB adapters are passed a DB Connection object (DbConFactory) - so connections are re-used and managed that way.
- $pdoDbHandler is the PDO database handler (connection) that will be used

Example:

```
\Qore\Factory\DbFactory::build(
   'PackName',
   'default',
   'stats',
   $c['bzstats.db.default.connection']);
```

if in your config.php you had: `$cfg['db']['default']['type']  = "mysql";`

DbFactory would try and return an instance of DbMysqlStats object from `<ROOT>/packs/PackName/dbadapters/dbmysqlstats.php`

The dbAapter should make sure that it gets an instance of an object which implements `\Qore\Unreal\iDB`.

This way it can use the `getInstance()` method to retrieve the PDO database handle that it will use:

```
namespace Packs\<PackName>\Dbadapters;

class DbMysqlTarget {
    ...

    /**
     * @param \Qore\Unreal\iDb $dbMysql
     */
    public function __construct(\Qore\Unreal\iDb $dbMysql) {
        $this->dbh = $dbMysql->getInstance();
    }
```

```
    ...
}
```

DbAdapters & Models

The way this all works is:

- db connection objects are registered in the D.I. container with IoC::Regsiter()
- DbAdapters are passed a db connection with IoC::Get()
  - this way connections can be re-used/shared between many db-adapters
  - db-adapters don't have to worry about opening/closing, just queries and that's it
  - each DbAdapter should implement an interface
  - the adapters should contain SQL *only* (just PDO calls to the DB, and return SQL to the model).
- models are passed a db-adapter, and use it to to talk to the underying DB
  - the constructor of the model should only accept the proper type of db-adapter
  - models then impose business logic/manipulate data as needed.
- models can extend \Qore\basemodel if needed - or you can roll your own model inheritance

This way, you can write model wrappers for your dbadapters that may impose additional logic to the returned SQL, but the dbadapters would be simple php files that query and return SQL only (no additional business logic).

This way, all you need to do to port to another database, is create a new dbadapter (dbpostgresstats.php), and a change in config.php

`$cfg['db']['default']['type']="postgres";`

and your wrappers would still be good, and nothing else changes in your app. Of course you will need to make sure that a corresponding dbcon object exists for your database. You can check Qore\dbcon folder to see or implement your own easily using Qore's DbConFactory.

Of Course, you don't need to use the dbAdapters/dbcon from Qore - if you prefer an ORM/ActiveRecord, then may I suggest you use doctrine, propell, or redbean.

Redbean is arguably easiest to implement but all 3 can easily be integrated by copying the php files under libraries, and including them in a basemodel class that all your

models will inherit from. Once that is done, you will instantly get ORM/ActiveRecord, and away you go.

Caching Full Pages & Sections

You can cache specific calls at the controller level - things like

SQL queries that are heavy, or any action that takes a while (loading something from a web service, etc..),

as well as cache complete page requests easily.

Every controller has a `$this->cache` object (which is an instance of Zend_Cache_Core), and you can use it to cache specific things in your controller:

```
    if ( ($data = $this->cache->load('my_unique_cache_id')) === false) {
        $data = $this->db->reallyLongQuery();
        $this->cache->save($data);
    }
```

This will cache `$data` for the amount of time specified in `$cfg['cache']['time']` in your `config/config.php` file.

On cache miss, it will load `$data` from whatever calls you need/make, and save it for the next request.

You can also cache an entire page if you want. To do this, you just need to use the controllers `$this->pageCache()` method.

as long as `$cfg['environment']` is not 'dev', then you can use this method, and it will generate the page cache for you.

This is extremely easy, and the only thing you have to do is to use `$this->cachePage()` in your public method's `__pre()`:

```
    public function myaction_public(array $args) {
        //do all the stuff your controller needs to display the page
        //nothing changes

        //create an array to hand the view with whatever data we need
        $data = array($args[0], $args[1]);

        //render the view as normal too - nothing changes
        echo $this->twig->render('myAction.html.twig', array('data' => $data));
    }
```

```
    public function myaction_public_pre() {
        //since this action is executed before myaction_public()
        //this either returns the cached page right away and exits
        //or executes the myaction_public() method - and renders the page
normally
        //and then caches the result for future re-use

        $this->cachePage();
    }
```

This will cache the full page for the amount of time defined in `$cfg['pageCache']['time']`.

If you don't like the idea of caching full pages at the controller/method level, you can cache pages in your `bootstrap.php` file centrally. Doing it this way is a bit faster as it needs to open less files - and the router isn't even loaded/processed, but means you have to manually specify the caching rules per URL. Although this does give you centralized control, and more control of the Zend_Cache object - the rule set may get lengthy on large sites.

See `bootstrap.php` for an example that is commented out. Just make sure that you do not use both the `bootstrap.php` method, and the controllers `$this->cachePage()` method - use 1 or the other! (or make sure that the there is no URL overlap between the `bootstrap.php` file, and the controller that you would like to use `$this->cachePage() from).

All caches (items level and page level) are stored on disk in `{QORE-ROOT}/tmp/cache`. Also please note that since `basecontroller.php` init's the cache objects, any services or non-view related controllers can use it.

Known issues

These are the issues that we are aware of:

Session Auto-Regeneration

Qore has a session auto-regeneration feature that can be useful for sites that wish to have a little more security. This way, the Qore can transparently regenerate the session id every N requests, which tries to make it more difficult to hijack a session.

This feature works great on simple pages, that is pages that only launch 1, or 2 HTTP Get request per browser refresh. On pages that launch more than 2 HTTP Get requests to controllers (perhaps through javascript), then this can become an issue. (please note that these are not XMLHttp calls, but pure HTTP Get requests from javascript. Qore will

already detect XMLHttp requests, and will not increment the auto-regeneration session counters for XMLHttp requests)

Consider the following:

We have a web page that has three (or more) sections. The “main content” HTML returned from the server (which is the Get request for the requested page), and 2 dynamic data sections which are loaded on the fly through Javascript calls which request data from other conrollers/methods that Qore serves up (perhaps some type of “api” for the site or something).

Now lets imagine that we tell Qore to auto-regeneration the session on every 4 requests. This produces the following order:

1. Initial Page load.
    1. session id set to “abc” - session cookie “abc” sent to browser.
        1. Auto-regeneration counter set to 1
    2. a javascript Http Get request is made for dynamic content retrieved from a controller/method.
        1. Browser sends “abc” session cookie
            1. Auto-regeneration counter set to 2
    3. a second javascript Http Get request is made for dynamic content retrieved from a controller/method.
        1. Browser sends “abc” session cookie
            1. Auto-regeneration counter set to 3

So far, everything is OK – but things get a little screwy when the user refreshes the page (or goes to another page which also has 2 or more “dynamic” content sections):

2. Second Page Request (user hits refresh on the same page – or goes to another page which also has multiple dynamic content sections).
    1. Browser sends “abc” session cookie
        1. Auto-regeneration counter set to 4 (the next request will regen the session)
        2. session id still “abc”
    2. a javascript Http Get request is made for dynamic content retrieved from a controller/method.
        1. Browser sends “abc” session cookie
        2. Auto-regeneration counter trigges session regeneration
            1. session id is regenerated to “def”
            2. session data is copied from “abc” into “def”
            3. session “abc” is deleted server side
            4. auto-regeneration counter set to 1 for session “def”

3. a second javascript Http Get request is made for dynamic content retrieved from a controller/method.
    1. Browser sends “abc” session cookie (oh ohh – the browser didn't get the new session cookie fast enough!!)
    2. session “abc” may not exist anymore, as the above session regenration may have deleted it (this is pure chance at this point)
        1. if “abc” session still exists – then things will sort-of be OK (user thinks there is nothing wrong):
            1. session regeneration happens:
                1. new session “ghi” is created
                2. session data “abc” is copied into “ghi”
                3. auto-regeneration counter set to 1 for session “ghi”
        2. if “abc” doesn't exist – then things are pretty bad for the user:
            1. “session “abc” doesn't exist!! Qore will kill all client-side cookies for the invalid session, and create a new blank/fresh session:
                1. new session “jkl” is created
                2. session auto-regeneration counter set to 1 for session “jkl”

so what happens next? Well, there are a couple of possibilities based on the chain of events above, but all aren't pretty:

- User gets Cookies back for sessions “def”, and “ghi”:

This one appears to work from the users perspective, as the user's session data exists in both, as the original “abc” session still existed when both new session were created. The problem is that the browser will choose one of these (the cookie that was last received) to use for the next request. As both will work, the user will be OK, and will continue to browse through the website unaffected, but on the server end, there will be an orphan session. This creates a bit of a security hole, as now there is a session that will stay there until the server-side cleans it up (the “max session lifetime” setting).

This is the best-case scenario, as the user isn't messed up, but on the server-side, it isn't great.

- User gets Cookies back for sessions “def” and “jkl”:

This is the worst one, and it's left 100% up to chance whether the user gets screwed or not.

In this case, session “def” is good, as the original “abc” session data was copied over into the new “def” session, but “jkl” is a new blank session (as “abc” was deleted before the 2nd dynamic content call, so Qore treated that as a “bad” session, and created a brand-new one).

If the browser chooses to use the “def” session cookie, then the user is OK, and “jkl” becomes an orphan (but because it's a new session – it isn't so bad).

If the browser chooses to use the “jkl” session cookie, then the user is screwed (the session is completely lost), and the “def” session becomes the orphan (which again opens up a security issue, as it will stick around until the session max lifetime cleanup).

So, what is the fix? Because this is a chance thing, there really isn't a good work-around that I have found..for now, if you need a page that has multiple dynamic content sections as outlined above, set Qore's auto-regeneration to 0 in your config.php to disable auto-regeneration. You can instead manually call *$this->session->regenerate();* in your controller at specific times where it makes sense (login, logout, etc..). this way you have control on when it happens, and can make the appropriate decisions so as not to create session orphans or screw the user (again, make sure that the page is pretty basic – at most 1 dynamic section (2 HTTP Get requests max).

- NOTE: Qore could simply wait a bit before really deleting sessions, but as this creates orphans and it has security implications, Qore will not implement this work-around.