Introduction

Welcome

RedBeanPHP is an easy-to-use, **on-the-fly** ORM for PHP. It's '**zero config**', relying on strict conventions instead. Let's look at the code, this is how you do **CRUD** in RedBeanPHP:

```
require 'rb.php';
R::setup();

$post = R::dispense('post');
$post->text = 'Hello World';

$id = R::store($post);       //Create or Update
$post = R::load('post',$id); //Retrieve
R::trash($post);             //Delete
```

This **automatically generates** the database, tables and columns... **on-the-fly**. It infers relations based on naming conventions. Download RedBeanPHP Now. RedBeanPHP is written by BDFL Gabor de Mooij and the RedBeanPHP community.

News

2014-06-29: RedBeanPHP 4.0.5 improved UUID/GUID support.
2014-06-01: Restructuring documentation
2014-05-27: RedBeanPHP 4.0.4 has been released (see below).
2014-05-21: Backport via-cache fix to 3.5.9
2014-05-18: RedBeanPHP 4.0.3 fixed issue with Via Cache.
2014-05-12: RedBeanPHP 4.0.2 fixed issue with nullifying parent bean.
2014-04-23: RedBeanPHP 4.0.1 link() now also accept camelcase type (converts to snake_case).
2014-04-06: RedBeanPHP 3.5 update 8 has been released, this is a maintenance update.
2014-04-01: We are proud to announce the official release of **RedBeanPHP 4.0**, right on time! (as always).

Zero Config

No need to configure anything. No **annotations** or lengthy **XML** files. Just follow the conventions and everything works.
Installation is also easy, just one file! No autoloaders, package management or include paths. RedBeanPHP also automatically configures your database

connection to use **UTF-8** encoding.

Fluid Schema

RedBeanPHP will adapt the database schema to fit your needs. It will create the tables and columns you need and change those columns if necessary.
During development, this will give you the '**NoSQL**' experience. However, when you deploy on a production server you can **freeze** the schema and benefit from performance gains and referential integrity. RedBeanPHP combines the best of both worlds: the **comfort** of **NoSQL** and the reliability of **relational database systems**.

Powerful

Although RedBeanPHP is a compact library, it's packed with powerful features. **FUSE** allows you to add **model** classes for records ad-hoc, as soon as RedBeanPHP discovers the presence of a domain model it will start using it. The duplicate method can create deep copies of any object structure in the database. The **Dispense** method can process entire arrays and convert them to object hierarchies. This is just a sample of all those powerful features offered by RedBeanPHP.

Download

Download the easy-to-use one-in-all package, one single file containing the entire RedBeanPHP library! **No composer**, **no auto-loaders**, **no configuration**, just download and run! Go to the download page and download to latest version of RedBeanPHP!

RedBeanPHP is a zero config, **fire-and-forget** ORM library, which is pretty fun, but it enforces strict database conventions. The best way to use RedBeanPHP is to start with an empty database and let RedBeanPHP build the schema for you **on-the-fly**. Don't try to *shoehorn* RedBeanPHP into existing projects with custom schemas.
Not sure if RedBeanPHP is the way to go ? Take a look at our checklist!

Restructuring documentation

At the moment, I am busy restructuring and improving the documentation. To improve this manual I might move some chapters around and reorganize some of the information. Sorry for the inconvenience.

Requirements

RedBeanPHP 4 requires PHP 5.3.4 or higher.If you are stuck with PHP 5.2 try RedBeanPHP 3.5. RedBeanPHP works on all well known operating systems. You need to have PDO installed and you need a PDO driver for the database you want to connect to. Most PHP stacks come with PDO and a bunch of drivers so this should not be a problem. RedBeanPHP supports a wide range of relational databases. RedBeanPHP also requires the MB String extension, once again, this is probably already there.

MySQL Strict Mode

RedBeanPHP does **not** work with **MySQL strict mode**. To turn off strict mode execute the following SQL query:

```
SET @@global.sql_mode= '';
```

Existing schemas

RedBeanPHP has been designed to build your database **on-the-fly**, as you go. Afterwards, you can **manually** change the schema to suit your needs (change column types, add additional indexes). Remember that the purpose of RedBeanPHP is to have an easy, configuration-less ORM. This can be achieved only by respecting certain conventions.

Ready to download?

Did you check your system requirements? Proceed to download RedBeanPHP.

Download

RedBeanPHP ships as a single, all-in-one **tarball**, that's all you need to get started with RedBeanPHP. Simply extract the **tar.gz** file to the destination folder and include the **rb.php** or **rb.phar** file in your PHP script.

Did you check the system requirements?

Download RedBeanPHP 4.0.5 recommended

The latest **stable** version of RedBeanPHP. If you're new to RedBeanPHP this is the recommended version to use.

Download RedBeanPHP

This version contains both a PHAR archive and a regular PHP file. This version also contains a **patch** for PHP 5.3.3 and earlier.

After downloading and extracting the file, include the rb-file in your PHP script.

If you use PHP 5.3.3 or older, run the P533 patch first.

Composer

RedBeanPHP does not support Composer because I have a more conservative approach to software distribution. However if you really want to use Composer, there is a simple workaround available. Thanks to Costin Moise for providing this workaround!

RedBeanPHP on Github

You can download RedBeanPHP from Github as well: Download RedBeanPHP from Github. On the Github page, look for a big button with the label 'download zip' and click on that button to download the source code of RedBeanPHP. After downloading run replica.php to assemble the all-in-one package.

Download RedBeanPHP 3.5 Legacy

This is the legacy version of RedBeanPHP (the **3.X series**). It will be supported until **1 October 2016**. Until then, I will answer questions and release updates and patches.

Download this version of RedBeanPHP if you are running an older version of PHP (< 5.3) or you rely on RedBeanPHP 3.5 specific features that have not been backported to the **4.X series**.

Download the latest version of RedBeanPHP legacy LTS: **RedBeanPHP 3.5.9**.

RedBeanPHP 4 Additional Plugin Pack

As of RedBeanPHP 4, the core package has been trimmed down. Some less essential components have been moved to this plugin package during the clean-up. The plugin package contains the following components:

- Support for CUBRID database for RedBeanPHP 4
- BeanCan Server Version 1 and 2 for RedBeanPHP 4
- Preloader (for eager loading features) for RedBeanPHP 4
- SQLHelper, a Query Builder for RedBeanPHP 4
- Old Cooker component for RedBeanPHP 4

Download the additional plugin pack for RedBeanPHP4.

Note that some of these plugins are considered deprecated. The **Cooker** and the **Preloader** have been replaced in RedBeanPHP 4 by different mechanisms. The **SQLHelper** is also considered deprecated. The **CUBRID** QueryWriter and the **BeanCan Servers** are still fully supported.

Next version of RedBeanPHP

For more details about the next version of RedBeanPHP please consult the RedBeanPHP roadmap. You can also download new beta versions on that page if you like to help us test the upcoming version of RedBeanPHP!

Old versions

It's still possible to download older versions of RedBeanPHP from the RedBeanPHP Download Archives. Feel free to visit the download archives and find one of the previous versions of RedBeanPHP.

RedBeanPHP does not support Composer or any other PHP package management system.

Install

Installing RedBeanPHP is very easy. Just download the RedBeanPHP all-in-one pack **TGZ** from the download page. After extracting the contents of the package, you will see the following files:

```
rb.phar
rb.php
p533patch.php
license.txt
```

You can either use the **PHAR** file or the regular **PHP** file. The rb.php file contains everything you need to start RedBeanPHP. Just include it in your PHP script like this:

```
    require 'rb.php';
```

If you prefer to use the PHAR file use:

```
    require 'rb.phar';
```

Simply include either the .phar or the .php and delete the one you don't use. You are now ready to use RedBeanPHP!
Let's try to setup a database connection!

Should I use PHAR or PHP?

Both the .phar file and the regular .php file provide the same functionality. Some IDEs do not fully support .phar files yet. Also, some **hosting** providers do not support .phar files because of configuration mistakes.
So, if you have some concerns about your hosting provider or your **IDE**, you can play safe by using the regular .php file.

Patch for PHP 5.3.3 and earlier

This patch is only required for people using old versions of PHP, i.e. PHP 5.3.3 or earlier. If you are running a PHP instance with a higher version number please skip this section. People using PHP version 5.3.3 or older should run the p533patch.php file first and then include the newly generated rb-p533.php file. The patch will modify the source for compatibility with these older PHP editions. Use the patch like this:

```
php p533patch.php
```

You will now see the following output:

```
Running Patch P533...
Applied patch for PHP < 5.3.3
```

After running the patch, you'll see a new file in your folder:

```
rb-p533.php
```

This is the file to be used with your version of PHP.

Connection

To connect to an **SQLite** testing database, without having to make one yourself, use:

```
    require 'rb.phar';
    R::setup();
```

On most systems, this just works.
This code creates a test database in your **/tmp** folder.
Of course, this is meant for testing purposes only (and to fool around), to connect to a real database, use one of the following snippets:

MariaDB

MariaDB (formerly known as **MySQL**) is the most popular database among web developers. Use MariaDB or MySQL for **light** web development. To connect to a MySQL database or a MariaDB database:

```
R::setup('mysql:host=localhost;dbname=mydatabase',
    'user','password'); //for both mysql or mariaDB
```

Did you manage to establish a connection to the database? Proceed to learn the basics of RedBeanPHP!

PostgreSQL

Postgres evolved from the classic Ingres database and is by far the most advanced database you can get. Use Postgres for serious application development. Postgres is rock solid and has lots of power features like window functions, support for hierarchical queries and materialized views. To connect to a PostgreSQL database:

```
R::setup('pgsql:host=localhost;dbname=mydatabase',
    'user','password');
```

SQLite

SQLite is file based database, ideal for embedded applications, prototyping, small (and smart) applications, small websites (not too much traffic) and data analysis. To connect to an SQLite database:

```
R::setup('sqlite:/tmp/dbfile.db');
```

CUBRID

CUBRID is an exciting database platform focusing on web development. It's an ideal replacement for rusty MySQL servers. While CUBRID seems to be almost completely compatible with MySQL it also offers a great deal of advanced features, like hierarchical queries and **click counters**. However, CUBRID also offers a very complete, easy-to-use GUI based toolchain. To use the CUBRID database with RedBeanPHP4, first install the plugin pack. To connect to a

CUBRID database:

```
R::setup('cubrid:host=localhost;port=30000;
dbname=mydatabase',
'user','password');
```

3rd Party DB support

Besides the databases mentioned on this page, additional support is provided by 3rd parties for: SQL Server and OCI databases. Please consult the git repository for further details. Interested in writing additional drivers? Please let me know!

Closing

To disconnect use:

```
R::close();
```

This will close the database connection.

Basics

CRUD

CRUD stands for **C**reate, **U**pdate, **R**etrieve and **D**elete. CRUD operations are the core of many web applications.

Working with beans

RedBeanPHP works with beans. Most interactions with the **database** are accomplished using beans. Beans are used to carry data from and to the database.

Every bean has a **type** and an **ID**. The type of a bean tells you which **table** in the database is used to store the bean. Every type maps to a corresponding table. The ID of a bean is the **primary key** of the corresponding **record**.
You can create a new bean by dispensing one.

Create

To create a new bean (of type 'book') use:

```
$book = R::dispense( 'book' );
```

You can now add properties:

```
$book->title = 'Learn to Program';
$book->rating = 10;
```

You can also use *array notation* if you like:

```
$book['price'] = 29.99; //you can use array notation as well
```

and store the bean in the database:

```
$id = R::store( $book );
```

At this point, the bean will be stored in the database and all **tables** and **columns** have been created.
The bean will now have an ID, which is also returned for your convenience.

RedBeanPHP will build all the necessary structures to store your data. However custom indexes and constraints have to be added manually (after freezing your web application).

Conventions

You can dispense any type of bean you like, as long as the type name consists of **lowercase alphabetical** characters:

```
$page = R::dispense('page'); //valid

$page = R::dispense( 'Page' ); //invalid: uppercase
$page = R::dispense( 'cms_page' ); //invalid: _
$page = R::dispense( '@#!' ); //invalid
```

However dispense also offers some shortcuts:

```
$twoBooks = R::dispense( 'book', 2 );

list($book, $page) = R::dispenseAll( 'book,page' );
list($book, $pages) = R::dispenseAll( 'book,page*2' );
```

Properties of beans may contain alphanumeric characters and underscores. **Camelcased** properties will automatically convert to **snake_case**:

```
$book->isSoldOut = true; //is_sold_out
$book->hasISBNCode = true; //has_isbn_code
```

Retrieve

To load a bean, simply pass the **type** and **ID** of the bean you're looking for:

```
$book = R::load( 'book', $id ); //reloads our book
```

If the bean does not exist an **empty** bean with ID **0** will be returned.

Update

To update a bean in the database, **add** or **change** properties:

```
$book->title = 'Learn to fly';
$book->rating = 'good';
$book->published = '2015-02-15';
R::store( $book );
```

Note that we added a new property 'published', RedBeanPHP will add a new column of type 'date' for this property. Also, it will widen the 'rating' from **INTEGER** to **VARCHAR** to support text as well as numbers.

```
//Examples of other data types
$meeting->when = '19:00:00'; //Time
$meeting->when = '1995-12-05'; //Date
$photo->created = '1995-12-05 19:00:00'; //Date time
$meeting->place = '(1,2)'; //only works in postgreSQL
```

You can use R::isoDate() and R::isoDateTime() to generate the current date(time) if you like.

As of RedBeanPHP 4.1 you can also use spatial columns for MySQL, learn more.

Delete

To delete a bean:

```
R::trash( $book ); //for one bean
R::trashAll( $books ); //for multiple beans
```

To delete all beans of a certain type:

```
R::wipe( 'book' ); //burns all the books!
```

To destroy the entire database simply invoke the nuclear method (be careful!):

```
R::nuke();
```

Batch

To load a series of beans use:

```
$books = R::loadAll( 'book', $ids );
```

Reload

To quickly reload a bean:

```
$bean = $bean->fresh();
```

Finding Beans

Instead of loading beans, you can also use the find() method to search for beans using certain criteria. Learn how to query beans in RedBeanPHP.

Finding

If you do not know the **ID** of a bean, you can search for beans using the **find** method:

```
$book  = R::find( 'book', ' rating > 4 ');
```

The find() method uses good old SQL. No fancy, custom query language — just plain old SQL.

The find operation in this example returns all beans of type **book** having a rating of four stars or more.

Find and SQL

The following example demonstrates how to use find() with bindings.

```
$books = R::find( 'book', ' title LIKE ? ', [ 'Learn to%' ] );
```

This find operation will return all beans of type 'book' having a title that begins with the phrase: 'Learn to'.

If find() has no results it will return an **empty array**.

There is no need to use mysql_real_escape. Always use the bindings.
Never use PHP variables in your query!

Find One

If you want a single bean instead of an array, use:

```
$book  = R::findOne( 'book', ' title = ? ', [ 'SQL Dreams' ]);
```

If no beans match the criteria, this function will return **NULL**.

Find All

Use **findAll** if you don't want to add any conditions:

```
$books = R::findAll( 'book' );
```

or if you just want to order or limit the results:

```
$book  = R::findOne( 'book', ' rating < 2 ');
```

If no beans match your criteria, this function returns an **empty array**.

Named slots

All find methods: find, findOne and findAll also accept named slots:

```
$books  = R::find( 'book', ' rating < :rating ', [ ':rating' => 2 ] );
```

Besides querying beans, you can also use regular SQL queries.

Querying

If you prefer **rows** rather than **beans** you may want to use the SQL query functions provided by **RedBeanPHP**. To execute a query:

```
R::exec( 'UPDATE page SET title="test" WHERE id=1' );
```

To get a **multidimensional** array:

```
R::getAll( 'SELECT * FROM page' );
```

The result of such a query will be a **multidimensional** array:

```
Array
(
    [0] => Array
        (
            [id] => 1
            [title] => frontpage
            [text] => hello
        )
    ...
)
```

Note that you can use **parameter bindings** as well:

```
R::getAll( 'SELECT * FROM page WHERE title = :title',
    [':title' => 'home']
);
```

To fetch a **single row**:

```
R::getRow( 'SELECT * FROM page WHERE title LIKE ? LIMIT 1',
    [ '%Jazz%' ]
);
```

To fetch a **single column**:

```
R::getCol( 'SELECT title FROM page' );
```

And finally, a **single cell**...

```
R::getCell( 'SELECT title FROM page LIMIT 1' );
```

To get an associative array with a specified key and value column use:

```
R::$adapter->getAssoc( 'SELECT id, title FROM page' );
```

In this case, the keys will be the IDs and the values will be the titles. **getAssocRow** will return complete rows.

Converting records to beans

You can convert rows to beans using the convertToBeans() function:

```
$sql = 'SELECT author.* FROM author
    JOIN club WHERE club.id = 7 ';
$rows = R::getAll( $sql );
$authors = R::convertToBeans( 'author', $rows );
```

Remember:
There is no need to use **mysql_real_escape** as long as you use parameter binding.

Besides querying you can also use other database functionality (like transactions) in RedBeanPHP. Learn more about database functions.

Database

This chapter discusses general database functionality of RedBeanPHP.

Reflection

To get all the columns of table '**book**':

```
$fields = R::inspect( 'book' );
```

To get all tables:

```
$listOfTables = R::inspect();
```

Multiple databases

There are two important methods to keep in mind when working with multiple **databases**: **addDatabase()** and **selectDatabase()**.
To add a new database connection use R::addDatabase() like this:

```
R::addDatabase( 'DB1', 'sqlite:/tmp/d1.db', 'usr', 'pss', $frozen );
```

To select a database, use the key you have previously specified:

```
R::selectDatabase( 'DB1' );
```

If you used **R::setup()** to connect to your database you can switch back to this database using:

```
R::selectDatabase( 'default' );
```

Transactions

RedBeanPHP offers three simple methods to use database **transactions**: begin(), commit() and rollback(). Usage:

```
R::begin();
try{
    R::store( $page );
    R::commit();
}
catch( Exception $e ) {
    R::rollback();
}
```

Because RedBeanPHP throws **exceptions**, you can catch the exceptions thrown by methods like R::store(), R::trash(), or one of your 'fuse' methods, and perform a rollback(). The rollback() will completely undo all the pending database changes.

Transaction closure

You can also use this variation:

```
R::transaction( function(){
    ..store some beans..
} );
```

The transaction() method supports nested transactions.

Note about auto-commits:
Many databases automatically commit after changing schemas, so make sure you test your transactions after **R::freeze(true);** !

Column Functions

As of RedBeanPHP 4.1 you can *bind* an **SQL function** to a column. This is useful for wrapping values when reading from / writing to the database.
For instance, to use *MySQL spatial data types* you need to prepare the columns like this:

```
R::bindFunc( 'read', 'location.point', 'asText' );
R::bindFunc( 'write', 'location.point', 'GeomFromText' );

$location = R::dispense( 'location' );
$location->point = 'POINT(14 6)';

//inserts using GeomFromText() function
R::store( $location );

//to unbind a function, pass NULL:
R::bindFunc( 'read', 'location.point', NULL );
```

While this method has been implemented to support MySQL spatial data types, you can use it for other purposes as well.
For instance, you can *encode* your own data types, create an **encryption** function or a **UUID** function.

As you have seen in the previous chapters RedBeanPHP will keep changing the schema to fit your needs, this is called '**fluid mode**'. While this is great for development, you don't want this to happen on your production server. Learn how to freeze your database for deployment.

Fluid and Frozen

RedBeanPHP has two modes, **fluid** and **frozen**.
As you have seen in the previous chapters, RedBeanPHP will keep changing the schema to fit your needs, this is what we call '**fluid mode**'.
While this is great for development you don't want this to happen on your production server. That's why you need to **freeze** the schema before you deploy

your application. To freeze your app, put **R::freeze(TRUE)** at the beginning of your script, like this:

```
require 'rb.php';
R::setup();
R::freeze( TRUE );
...your code...
```

After freezing the schema, open your database client and inspect the schema RedBeanPHP has created for you. You can now refine it where necessary, i.e. add some indexes, change some columns, add or change constraints and more.

Always review the schema generated by RedBeanPHP and allow yourself some time to refine it.
Do not change the table or column names though, these are part of the RedBeanPHP conventions. Instead I recommend to inspect and refine column types (maybe a bit too wide ?), indexes and foreign key constraint settings.

Partial freezing (Chill mode)

It's also possible to only lock the schemas of several types of beans, for instance to lock the schemas of bean types 'book', 'page' and 'book_page' use:

```
R::freeze( ['book','page','book_page'] );
```

This makes RedBeanPHP operate in fluid mode, but those tables will not get modified. To reset, pass an empty array.

Learn how to use the debugging tools.

Debugging

The debug() method will reveal all queries being executed by RedBeanPHP:

```
//turns debugging ON
R::debug( TRUE );

//turns debugging OFF
R::debug( FALSE );
```

The queries will be printed on the screen. The output of the debugging function looks like this:

```
INSERT INTO "event" ( id, "name" ) VALUES
( NULL, ? )
Array
```

```
(
[0] => party
)
```

You can also log the queries:

```
R::debug( TRUE, 1 ); //select mode 1 to suppress screen output
```

To access the logs:

```
$logs = R::getDatabaseAdapter()
        ->getDatabase()
        ->getLogger();

print_r( $logs->grep( 'SELECT' ) );
```

Use the grep() method to search the logs.

Query parameters

By default, the debugger prints the queries and parameters in separate sections. Sometimes you might prefer to see what the actual query would look like if the parameters had been filled in. RedBeanPHP 4.1+ offers two new debugger modes to facilitate this:

```
R::debug(true, 2); //select MODE 2 to see parameters filled in
```

Outputs the query above like this:

```
INSERT INTO "event" ( id, "name" ) VALUES
( NULL, "party" )
```

Mode 2 also writes to the logs, if you want to suppress screen output, select mode 3.

Under the hood

Under the hood, all debugging functionality makes use of the logger classes. There are two logger classes available in RedBeanPHP: the Default Logger and the Debugger Logger (4.1+). Besides using the convenience methods listed here you can create your own logger instance and attach it to some object:

```
$myLogger = new \RedBeanPHP\Logger\RDefault;
$database->setLogger($myLogger);
```

Inspecting Beans

The easiest way to inspect a bean is to just echo it.

```
echo $bean;
```

If you have a list of beans, an array, you can use good old print_r of course, but print_r will also print useless details. To get a shorter and more descriptive summary of a bean or an array of beans you can use the dump() function:

```
print_r( R::dump($myBeans) );
print_r( R::dump($singleBean) );
```

The output looks like this:

```
[1] => {"id":"1","name":"party"}
```

An even shorter syntax:

```
dmp( $myBean );
```

The dmp() function is a global function for your convenience.

Error handling

Error handling is different in fluid and frozen mode. In fluid mode SQL errors caused by missing columns or tables will be suppressed but other errors (syntax) will throw **RedException\SQL** exceptions.

SQLite and some plugin drivers do not provide meaningful **SQLSTATE** codes, therefore under these drivers fluid mode will suppress all errors.

Testing the connection

In RedBeanPHP 4.1+ you can test the connection using a special test function. This function will refrain from throwing exceptions and simply return TRUE if the connection has been established and FALSE otherwise:

```
$isConnected = R::testConnection();
```

Besides debugging, this function is handy for installers and setup scripts of web applications to determine whether database credentials have been entered correctly.

Learn about relations in RedBeanPHP.

Relations

One-to-many

In a **one-to-many** relation, one bean has a list of other beans but all those beans cannot belong to another bean at the same time. For instance, let's create a shop:

```
$shop = R::dispense( 'shop' );
$shop->name = 'Antiques';
```

To add products to the shop, add beans to the **ownProductList** property, like this:

```
$vase = R::dispense( 'product' );
$vase->price = 25;
$shop->ownProductList[] = $vase
R::store( $shop );
```

Each product in the **ownProductList** belongs to shop and *cannot* belong to another shop.

Note that the name of the list **has to match the type of beans it contains**. So, the 'ownProductList' contains beans of type 'product', a pageList contains pages, an 'ownCarList' contains 'cars' and so on. This convention is used to create the database mapping, in case of the shop, every product record will get a **'shop_id'** field.

When you access an own-list, RedBeanPHP will query the related beans and populate the array, this is called *lazy loading*. So, to load the list:

```
$shop = R::load( 'shop', $id );
$first = reset( $shop->ownProductList ); //gets first product
$last = end( $shop->ownProductList ); //gets last product
foreach( $shop->ownProductList as $product ) {...} //iterate
```

To remove the products from the shop:

```
//remove one product by its ID
unset( $store->ownProductList[$id] );

//remove all
$store->ownProductList = array();
R::store( $shop );
```

To replace the current list of products:

```
$store->ownProductList = array( $vase, $lamp );
```

Exclusive mode

Note that those products continue to *exist* in the database, they are just **unrelated**, don't want that ? Then open the **own-list** in **exclusive mode**, using the **x-own-list** like this:

```
$shop->xownProductList = array();
R::store( $shop );
```

Emptying the list now will cause the vases to be gone too. In *exclusive mode* the beans in the list are considered to be dependent on their owner. If they are removed from the list, they are deleted as well (i.e. they **depend** exclusively on their owner).

When using the **x-own-list** from the **start**, if you **delete** the shop, its products will be **deleted** as well.

This happens because the first time you access an own-list, a foreign key will be created for the owned bean, one that will **CASCADE ON DELETE** for an x-own-list and one that will **SET-TO-NULL** otherwise. Once the foreign key is in place, it will not be modified by RedBeanPHP anymore. However you can always change the constraint manually using your database client.

Other end of the one-to-many

The other end of the one-to-many relation is the many-to-one relation. Learn more about the many-to-one relation.

Many-to-one

Now let's look at this relation from the perspective of a product. A product belongs to a shop, so you can access the shop like this:

```
$shop = $product->shop;
```

This is called the **'parent bean'**.
The shop is considered the parent of the product. It **owns** the product.

Removing the parent

To remove the shop from our product in the example above, simply assign the value **NULL** to the property 'shop':

```
$product->shop = NULL; //removes product from shop
```

Besides one-to-many, RedBeanPHP has a special version of this relation: the one-to-X also known as the one-to-fixed relation. Read more about the One-to-fixed relation.

One-to-Fixed

Sometimes you want to refer to a bean using a different name. For instance, when you have a course referring to a teacher and a student, both of which are people. In this case you can use **fetchAs**:

```
$c = R::dispense( 'course' );

//At assignment time, no difference...
$c->teacher = R::dispense( 'person' );
$c->student = R::dispense( 'person' );

$id = R::store( $c );
$c = R::load( 'course', $id );

//when accessing the aliased properties,
//tell RedBeanPHP how to find the bean:
$teacher = $c->fetchAs('person')->teacher;
```

fetchAs tells RedBeanPHP the **ID** has to be associated with a different *type* (in this case 'person' instead of 'teacher' or 'student'). This also works the other way:

```
//returns all courses for this person
//where he/she is the teacher.
$person->alias('teacher')->ownCourseList;
```

From a relational point of view, we have **exactly** two people for every row (although one or both can be NULL of course). This is why we call these 'aliases' **one-to-X** relations (or **one-to-fixed** relations), where *X* is a fixed number.
You can use as many of these 'aliases' as you like.

Also learn about Many-to-many relations.

Many-to-many

A **shared list** contains beans that may be associated with more than just one other bean (**many-to-many relation**). Tags are a common example:

```
list($vase, $lamp) = R::dispense('product', 2);

$tag = R::dispense( 'tag' );
$tag->name = 'Art Deco';
```

```
//creates product_tag table!
$vase->sharedTagList[] = $tag;
$lamp->sharedTagList[] = $tag;
R::storeAll( [$vase, $lamp] );
```

In this example, a product can have multiple tags and every tag in the list can be associated with other products as well. The latter was *not* possible in the one-to-many relation.

Like the **own-list** the name of the **shared-list** has to **match the type** of beans it contains. In the database, these assocations will be stored using a link table called 'product_tag'.

This link table is cleaned up automatically, if you break the association between two beans in a shared list the link record is removed as well. Also note that a shared list cannot have **aliases** and always applies a UNIQUE constraint (you cannot have duplicate links). In some situations this means you have to use a slighly different approach; the N11N relation.

Via relations

Using the **via()** method, you can treat normal beans as if they were **N-M** relations:

```
$participant->project = $project;
$participant->employee = $lisa;
$participant->role = 'developer';
R::store( $participant );

//get all associated employees via the participants
//(includes $lisa!)
$employees = $project
    ->via( 'participant' )
    ->sharedEmployeeList;
```

Remember that, since unrelated link beans are removed automatically, emptying a shared list (even using via) causes the link beans to be removed! However, you can always *nullify* the relations manually of course.

Via is **sticky**, once you tell RedBeanPHP to fetch a type of bean via another bean it will remember this for the rest of the program. So, once you told RedBeanPHP: $project->via('participant')->sharedEmployee it will always load employees using the participant table as a link table, even if you later say: $project->sharedEmployee. Also note that via() reloads the list.

Self referential N-M

You can have a shared list containing beans of the same type as the owner of the list:

```
$friends = $friend->sharedFriend;
```

In this case RedBeanPHP will operate in a special self-referential many-to-many relationship mode. It will not only retrieve all friends of $friend, but also all other friends that are associated with $friend.

You can use two complimentary one-to-many relations as one many-to-many relation. This is called an aggregation or N11N-relation.

Using SQL Snippets

You can modify the contents of an own-list and a shared-list using additional **SQL snippets**. Use **with()** to order or limit the list and **withCondition** to add additional filtering.

```
$pages = $book
    ->with( ' ORDER BY pagenum ASC ' )
    ->ownPageList;

$vases = $shop
    ->withCondition(' category = ? ', ['vase'] )
    ->ownProductList;

//combine condition and order
$vases = $shop
    ->withCondition(' category = ? ORDER BY price ASC ', ['vase'] )
    ->ownProductList;

$employees = $project
    ->withCondition(' priority > 40 ')
    ->sharedProjectList;

//Special case, filter on linking records...
$employees = $project
    ->withCondition(' employee_project.assigned < ? ', [ $date ])
    ->sharedProjectList;
```

Note the last case in this example. Here we use a column from the link table to filter the rows. This technique allows you to filter on relational qualifications like the duration of the assignment to the project.

You cannot combine **with()** and **withCondition()**. Instead, you can append additional clauses like in the third example.

Via and SQL

Via can be used with SQL snippets as well:

```
$designers = $project
    ->withCondition(' participant.role = ? ', ['designer'] )
    ->via( 'participant' )
    ->sharedEmployeeList;
```

Both with() and withCondition() cause the list to reload, however if the SQL snippet hasn't changed and the writer cache is active (default) then no query will be send to the database.

Reloading a list

To reload a list without an SQL snippet use the **all()** method or unset it:

```
$shop->all()->ownProductList;
unset( $shop->ownProductList ); //will be reloaded next time.
```

The noLoad modifier

Sometimes, when you only want to add something to a list, there is no need to load the entire list. To keep RedBeanPHP from loading a list use the noLoad modifier as depicted in the following example:

```
$book->noLoad()->xownPageList[] = $newPage;
```

This will add a new page to the list, but the initial loading of the list will not take place.

Counting

Counting records is very easy with RedBeanPHP. For instance, to count all beans of type *book* use:

```
$numOfBooks = R::count( 'book' );
```

You can use additional SQL here as well:

```
$numOfBooks = R::count( 'book', ' pages > ? ', [ 250 ] );
```

Count related beans

Counting related beans is just as simple. To count all the pages of a 'book' bean:

```
$numPages = $book->countOwn( 'page' );
```

You can use the same technique for shared lists:

```
$numProjects = $member->countShared( 'project' );
```

You can also use withCondition() and alias():

```
$numProj = $member
        ->withCondition(' member_project.role ', ['lead'] ) )
        ->countShared( 'project' );

$numPages = $book
        ->withCondition( ' book_page.number > ? ', [100] )
        ->countOwn( 'page' );

$andy->alias( 'coAuthor' )->countOwn( 'book' );

$shop->via( 'relation' )->countShared( 'customer' );
```

The first example counts all projects associated with the member in which the member has the 'lead' role. The second example counts all the pages of a book having a page number > 100. Finally the last example demonstrates the use of an aliased list. Here we count the number of books written by Andy where he has been the co-author. All count operations return a number.

Labels, Enums, Tags

Labels, Enums and Tags are all based on very simple beans. These beans only have an **id**, a **type** and a **name**. While they might look simple, these beans can offer various powerful services in your applications. Labels form the basis for enums and tags. Enums are in fact labels in a one-to-many relation while Tags are labels in a many-to-many relation.

Labels

A Label is a bean with just a name property. You can generate a batch of labels of a certain type using:

```
$labels = R::dispenseLabels( 'meals', ['pizza', 'pasta']);
```

This will create two meal objects. Each bean will have a name property that corresponds to one of the strings in array.

You can also collect the strings from label beans using:

```
$array = R::gatherLabels( $meals );
```

The gatherLabels() function returns an alphabetically sorted array of strings each containing one name property of a bean in the bean list provided.

Enums

An enum type is a special bean that enables for a property to be a set of predefined values. To use an ENUM:

```
$tea->flavour = R::enum( 'flavour:english' );
```

The ENUM method will do a lot of work here. First it checks whether there exists a 'flavour' bean with the name 'ENGLISH'. If this is the case, enum() will return this bean, otherwise it will create such a bean, store it in the database and return it. This way your ENUMs are created on the fly - properly. To compare an enum value:

```
$tea->flavour->equals( R::enum( 'flavour:english' ) );
```

To get a list of all flavours, just omit the value part:

```
$flavours = R::enum( 'flavour' );
```

To get a comma separated list of flavours you might want to combine this method with other Label Maker methods:

```
implode( ',', R::gatherLabels( R::enum( 'flavour' ) ) );
```

Since RedBeanPHP enums are beans you can add other properties as well. To query using an enum:

```
$flowers = R::find( 'flower', ' color_id = ? ', [ R::enum( 'color:red' )->id ] );
```

The find query above will retrieve all red flowers. While this query is perfectly readable the syntax is a bit clunky, therefore there is a shorthand notation for the R::enum(...)->id part:

```
$flowers = R::find( 'flower', ' color_id = ? ', [ EID('color:red') ] );
```

The global function EID() returns the ID of the given ENUM directly.

Tags

Tags are often used to categorize or group items. To tag a an item:

```
R::tag( $page, array( 'topsecret', 'mi6' ) );
```

To **fetch all tags** attached to a certain bean we use the same method but without the tag parameter:

```
$tags = R::tag( $page ); //returns array with tags
```

To **untag** an item use:

```
R::untag( $bean, $tagListArray );
```

To get all beans that have been tagged with $tags, use **tagged()**:

```
R::tagged( $beanType, $tagList );
```

To find out whether beans have been tagged with specific tags, use **hasTag()**:

```
R::hasTag($bean, $tags, $all=false)
```

To **add tags** without removing the old ones:

```
R::addTags( $page, array('funny', 'hilarious') );
```

To get beans that have ALL these tags:

```
//must be tagged with both tags
R::taggedAll( $page, ['funny', 'hilarious'] );
```

Advanced

Trees

RedBeanPHP supports **self-referential** relationships. In RedBeanPHP terminology, these are called trees. Here is an example, let's decorate a christmas tree with some candy canes:

```
$cane = R::dispense('cane',10);
$cane[1]->ownCane = array( $cane[2], $cane[9] );
$cane[2]->ownCane = array( $cane[3], $cane[4] );
$cane[4]->ownCane = array( $cane[5],
            $cane[7], $cane[8] );
$cane[5]->ownCane = array( $cane[6] );
$id = R::store($cane[1]);
$root = R::load('cane',$id);

echo $root->ownCane[2]->ownCane[4]
    ->ownCane[5]->ownCane[6]->id;
//outputs: 6
```

Trees are just a special case of lists, you use a list with the same name as the parent type. In the example script above, a **cane** has an **ownCaneList**. Another example: *page->ownPageList*. As you can see in the example above you can navigate the lists using the IDs.

Traversal

Instead of manually looping through each own-list of a bean you can use the traverse() method:

```
$page->traverse( 'ownPage', function( $page ) {
    ....
} );
```

This allows you to recursively apply a function to a list. To limit the results when accessing a list you can use the with/withCondition() method:

```
$page->with( ' LIMIT 10 ')->traverse( ... );
$page->withCondition( '  rating > ? ', [ 5 ] )->traverse( ... );
```

You can also use **withCondition** and **alias** together with the traverse function.

Use the third parameter to specify the maximum depth:

```
$page->traverse( 'ownPage', $func, 3 ); //max 3 levels
```

Use the PHP **use** statement to import variables into the function scope:

```
$task->traverse( 'ownTask', function( $task ) use ( &$todos ) {
    $todos[] = $task->name;
} );
```

The traverse() function does not check for *recursion* in trees.

Link Beans

The following code associates an employee with a project using a many-to-many relation.

```
$project->sharedEmployeeList[] = $employee;
```

The project and the employee are linked by a **link bean**. Link beans have their own **type**, in this case: **employeeProject**. In the database, these beans are stored in the **employee_project** table.

Qualified links

Sometimes you want to qualify a relationship. For instance, in the case of projects and employees, you might want to add a 'role' property to the relation:

```
list($e, $p) = R::dispenseAll('employee,project');
$p->link( 'employee_project', [
    'role' => 'director'
] )->project = $p;
```

While this is quite handy, it's often better to introduce the missing concept: **participant** in this case. Often, when you find yourself qualifying a relationship, you might have missed an important part of your data model. A relation or link is not a very good substitute for this.

Be careful with adding to much properties and logic to relations, make sure you haven't missed an important concept in your domain model.

Accessing link beans

Since a many-to-many relation can be viewed as a combination of two one-to-many relations you can access the link beans through the ownList on either side of the relation. In the case of a project-employee relations you can access the intermediate bean like this:

```
$employee->ownEmployeeProjectList;
```

To remove the intermediate beans upon assigning an empty array open the list in exclusive mode:

```
$employee->xownEmployeeProjectList;
```

Other relations

This chapter discusses less common relations.

Aggregations

It's possible to treat two complementary **N-1** relations as one many-to-many relation, thus benefiting from the advantages of a shared list. In **RedBeanPHP 4.1+** you can use the **aggr()** method of a bean to collect the parent beans for each member of an own-list:

```
$targets = $quest1->aggr( 'ownQuestTargetList', 'target', 'quest' );
```

This code will iterate over the ownQuestionTargetList and for every questTarget bean in the list it will load the bean in the target property as a bean of type 'quest'. This relation could not have been formed as as shared list because a shared list does not allow aliases. Without aliases the relation would have been a *symmetrical* one, lacking the notion of direction. Another solution to this problem would be to use the shared list and create a **VIEW** of quest called target.

One-to-one

One-to-one relations are not used frequently. Traditional 1-1 records are linked by their primary keys. Load them like this:

```
list( $author, $bio ) = R::loadMulti( 'author,bio', $id );
```

This loads an author and a biography with the same ID. You need to make sure the IDs are in sync yourself.

In RedBeanPHP one-to-one relations are an **anti-pattern**, the fields should belong to the same bean. This method has been added for compatibility reasons only, try to avoid it!

Polymorph relations

To load a bean whose type is determined by another column:

```
$ad = $page->poly( 'contentType' )->content;
```

This code returns the bean referred to in **content_id** using the bean type specified in column **content_type**. If content_type contains the value 'advertisement' the content will be a bean of type 'advertisement'.

This is an **anti-pattern** in RedBeanPHP, do not use this functionality unless you have to.
Use poly() to retrieve polymorph data from an external or legacy database only.

Models

A **model** is a place to put validation and business logic. Imagine a Jazz band that can only have up to 4 members. We could implement this rule like this:

```
if ( count( $members ) > 4 )
throw new Exception( 'Too many!' );

$band->ownMember = $members;
R::store( $band );
```

However, now we need to add this check *everytime* we call R::store(). It would be much more convenient if R::store() was smart enough to perform this check by itself. We can accomplish this by putting the **validation rule** in our model. RedBeanPHP automatically discovers the models that belong to beans, so we can implement this validation like this:

```
class Model_Band extends RedBean_SimpleModel {
        public function update() {
            if ( count( $this->bean->ownMember ) >4 )
            throw new Exception( 'Too many members!' );
        }
}

list( $band, $members ) = R::dispenseAll( 'band,member*5' );
$band->ownMember = $members;
R::store( $band ); //will trigger exception
```

RedBeanPHP automatically connects beans with models using a naming convention (i.e. Model_{TYPE OF BEAN}).
Now, every time we call store and something is wrong with the number of members, an exception will be triggered automatically. The mechanism that connects beans to models is called **FUSE**, because beans are *fused* with their models. Within a model, **$this->bean** refers to the bean.
To create a model for your bean, simply add a class like this:

```
//with classic namespace style
class Model_Band extends RedBean_SimpleModel { ... }
```

If you like your models to reside in the namespace **\Model**, you can set the following constant:

```
//with namespace Model
define('REDBEAN_MODEL_PREFIX', '\\Model\\')
```

You can now create a model class like this:

```
class \Model\Band extends \RedBeanPHP\SimpleModel { ... }
```

If you prefer no namespacing at all:

```
//use plain classes (without any namespacing)
define('REDBEAN_MODEL_PREFIX', '')

class Band extends \RedBeanPHP\SimpleModel { ... }
```

Beans of types like 'book_page' will search for a model 'BookPage' first and if no such model is found they will try to connect to 'Book_Page'.

Scoping rules

Within the model, the **$this->bean** variable refers to the bean. Simply **$this** also refers to the bean but without returning references, in practice this can be very confusing so I recommend to use **$this->bean**.

Fused methods

Besides **update()** RedBeanPHP FUSE calls other methods on the model as well:
R::store() invokes **update()** and **after_update()**,
R::load() invokes **open()**,
R::trash() invokes **delete()** and **after_delete()**,
R::dispense() invokes **dispense()**.

Note that since loading a bean also causes a new bean to be dispensed to receive the record from the database, load also invokes **dispense()**.

Example: all fused methods

To demonstrate the order and use of all of these methods let's consider an example:

```
$lifeCycle = '';
class Model_Bandmember extends RedBean_SimpleModel {
    public function open() {
       global $lifeCycle;
       $lifeCycle .= "called open: ".$this->id;
    }
    public function dispense() {
        global $lifeCycle;
        $lifeCycle .= "called dispense() ".$this->bean;
    }
    public function update() {
        global $lifeCycle;
        $lifeCycle .= "called update() ".$this->bean;
    }
    public function after_update() {
        global $lifeCycle;
        $lifeCycle .= "called after_update() ".$this->bean;
    }
    public function delete() {
        global $lifeCycle;
        $lifeCycle .= "called delete() ".$this->bean;
    }
    public function after_delete() {
        global $lifeCycle;
        $lifeCycle .= "called after_delete() ".$this->bean;
    }
}

$bandmember = R::dispense( 'bandmember' );
$bandmember->name = 'Fatz Waller';
$id = R::store( $bandmember );
$bandmember = R::load( 'bandmember', $id );

R::trash( $bandmember );
echo $lifeCycle;
```

output:

```
    called dispense() {"id":0}
    called update() {"id":0,"name":"Fatz Waller"}
    called after_update() {"id":5,"name":"Fatz Waller"}
    called dispense() {"id":0}
    called open: 5
    called delete() {"id":"5","band_id":null,"name":"Fatz Waller"}
    called after_delete() {"id":0,"band_id":null,"name":"Fatz Waller"}
```

Custom FUSED methods

Besides the standard methods mentioned above, any method on the model can be invoked by calling it on the bean (assuming it does not collide with a native bean method):

```
    $dog = R::dispense( 'dog' );

    //call bark() on Model_Dog:
    $dog->bark();
```

Boxing and Unboxing

If you have a bean and you want to obtain the corresponding model use:

```
    $dogBean = R::dispense( 'dog' );

    //get reference to Model_Dog
    $dogModel = $dogBean->box();
```

Similarly, if you have a model and you want its inner bean, call:

```
    $dogBean = $dogModel->unbox();
```

We call this technique **boxing** (and unboxing). This can be handy if you want to make use of typehinting:

```
    public function addDog( Model_Dog $dog ) {
        ...
    }
```

Otherwise, we would have to use type RedBean_OODBBean which is less descriptive.

Model Factory and Dependency Injection

If for some reason you need to control how the bean turns into a model you can pass a factory function like this:

```
use RedBeanPHP\BeanHelper\SimpleFacadeBeanHelper as SimpleFacadeBeanHelper;
SimpleFacadeBeanHelper::setFactoryFunction( function( $beanTypeName ) {
    $model = new $name();
    $model->setMailer( new MailLib() );
    return $model;
} );
```

In this example we inject a mail library in a model using the factory function. For more complex scenarios you can even use the factory to pass the model to your own dependency injection framework.

Meta data

Beans contain meta data. For instance, the **type** of the bean is stored in the meta data. To obtain the type of a bean:

```
$bean->getMeta('type');
```

You can also store your own meta data in a bean:

```
$bean->setMeta( 'my.secret.property', 'secret' );
```

this data will not get stored in the database.

Tainted

Some meta data is accessible using convenience method. For instance if you would like to know whether a bean has been changed since it got retrieved from the database use the tainted() method.

```
$bean->isTainted();

//or:

$bean->getMeta('tainted');
```

Note that a bean is marked as tainted if a list gets **accessed**. You can also set the tainted flag yourself.

Old

To determine if a certain property has changed:

```
$book = R::load( 'book', $id );
$book->hasChanged( 'title' ); //returns FALSE
$book->title = 'New title';
$book->hasChanged( 'title' ); //returns TRUE
```

To get the old value of the property:

```
$book->old( 'title' );
```

Testing Equality

To test whether two beans have the same type and **primary key** ID:

```
$bean->equals( $otherBean );
```

Empty

To determine if a bean is empty, or only contains **empty** values (everything that qualifies as empty() in PHP) use:

```
$bean->isEmpty();
```

Copy meta data

You can copy meta data from another bean like this:

```
$bean->copyMetaFrom( $otherBean );
```

Duplicate

R::dup() makes a deep copy of a bean properly and without storing the bean. All beans in own-lists will be duplicated recursively. All references to shared beans will be copied but not the shared beans themselves. All references to parent objects (_id fields) will be copied but not the parents themselves. The bean will not be stored so you have the chance to modify it before saving. Usage:

```
//entire bean hierarchy
$book->sharedReader[] = $reader;
$book->ownPage[] =$page;
$duplicated = R::dup( $book );
```

```
//..change something...
$book->name = 'copy!';
//..then store...
R::store( $duplicated );
```

As of **RedBeanPHP 4**, the R::dup() method also duplicates trees, in earlier versions the duplication manager skipped tree lists ($page->ownPage). Note that duplication/export won't work for aliased beans.

Performance

Both dup() and exportAll() need to query the database schema which is slow. To speed up the process you can pass a database schema:

```
R::$duplicationManager->setTables( $schema );
```

To obtain the schema use:

```
$schema = R::$duplicationManager->getSchema();
```

You can now use this schema to feed it to setTables(). R::dup() and R::exportAll() both use this schema.

Import and Export

RedBeanPHP offers several functions to exchange data with beans.

Import

You can import an array into a bean using:

```
$book->import($_POST);
```

The code above is handy if your $_POST request array only contains book data. It will simply load all data into the book bean. You can also add a selection filter:

```
$book->import($_POST, 'title,subtitle,summary,price');
```

This will restrict the import to the fields specified. Note that this does not apply any form of **validation** to the bean. Validation rules have to be written in the *model* or the *controller*.

To import from another bean:

```
$book->importFrom( $otherBean );
```

Import using Dispense

Dispense can even convert a multi dimensional array to a bean hierarchy like this (use _type to indicate the type of the bean):

```
$book = R::dispense( [
    '_type' => 'book',
    'title'  => 'Gifted Programmers',
    'author' => [ '_type' => 'author', 'name' => 'Xavier' ],
    'ownPageList' => [ ['_type'=>'page', 'text' => '...'] ]
] );
```

Export

To export the properties and values of a single bean use:

```
$array = $bean->export();
```

To recursively export one or an array of beans use:

```
$arrays = R::exportAll( $beans );
```

Bean lists in exports are keyless, 0 indexed. To also export parent beans:

```
$arrays = R::exportAll( $beans, TRUE );
```

Non-static

If you don't like static methods, you can use the objects behind the facade directly. Almost every method of the R-class is available through the original RedBeanPHP objects as well. The facade is just that: a thin layer on top of these objects. Here is an overview of the most important R-methods and how to use them 'the non-static way'.

Note that there are three important objects in RedBeanPHP: the adapter (DBAdapter), the query writer (QueryWriter) and the RedBeanPHP object database (OODB). We call these objects the core objects, because together they represent the foundation of RedBeanPHP. Other objects need these core objects, that's why they are bundled in a toolbox (ToolBox). So, if you need let's say an instance of the Tag Manager class (TagManager) you'll have to pass an instance of the toolbox to the contructor.

Toolbox

You can manually assemble your toolbox like this:

```
$pdo = new RPDO($dsn);
$adapter = new DBAdapter($pdo);
$writer = new MySQL($adapter);
$oodb = new OODB($writer);
$tb = new ToolBox($oodb, $adapter, $writer);
```

Wiring

RedBeanPHP has a very decoupled architecture, which makes it very flexibile. However this means you need to introduce some objects to eachother. First we need to tell RedBeanPHP how beans can obtain the toolbox, this means we need to define our own BeanHelper:

```
class BeanHelper extends BeanHelperFacade {
        private $toolbox;
        public function getToolbox() {
                return $this->toolbox;
        }
        public function setToolbox($toolbox) {
                $this->toolbox = $toolbox;
        }
}
```

Now let's do the wiring:

```
$r = $tb->getRedBean();

//A helper for OODB to give to its beans
$b = new BeanHelper;
$b->setToolbox($tb);
$r->setBeanHelper($b);

//allow OODB to associate beans
$r->setAssociationManager(new AssociationManager($tb));

//enable FUSE
$h = new SimpleModelHelper;
$h->attachEventListeners($r);
```

Hybrid

Normally the facade does all this dull work for you. You can also let the facade do this work and still work with instances; simply steal the toolbox from the facade after it has been configured:

```
R::setup(...);
$toolbox = R::getToolBox(); //give it to me!
```

Service objects

Many methods in the R-facade are just *wrappers* around calls to methods on one of these core objects: **OODB**, **Writer** and **Adapter**. However many static methods in R also call so-called service objects. Service objects offer secondary functionality. To instantiate a *service object* you need to pass the toolbox to its constructor. The toolbox contains everything service object needs to operate: the adapter to connect to the database, the OODB object to call basic ORM methods and the writer to write queries for the database.

For instance, R::find() uses the Finder class. To create an instance of Finder yourself:

```
$f = new Finder($tb);
```

That's it. Now we have an instance of the Finder service object. Now to find a bean use:

```
$x = $f->find( 'music', ' composer = ? ', 'Bach' );
```

API

This manual focuses on the facade. For details on individual objects, please consult the API pages.

Read more about the internals of RedBeanPHP.

UUIDs

RedBeanPHP has not been designed for use with UUIDs or GUIDs. However if you really want, you can tune RedBeanPHP to support this.

Enabling UUID support in MySQL

To enable **UUID** support in MySQL in fluid and frozen mode you need to provide your own *QueryWriter*. Here is an example of a QueryWriter that enables UUIDs in MySQL:

```
class UUIDWriterMySQL extends MySQL {

    protected $defaultValue = '@uuid';
    const C_DATATYPE_SPECIAL_UUID  = 97;

    public function __construct( Adapter $adapter ) {
        parent::__construct( $adapter );
        $this->addDataType(
        self::C_DATATYPE_SPECIAL_UUID, 'char(36)'  );
    }

    public function createTable( $table ) {
        $table = $this->esc( $table );
        $sql   = "
            CREATE TABLE {$table} (
            id char(36) NOT NULL,
            PRIMARY KEY ( id ))
            ENGINE = InnoDB DEFAULT
            CHARSET=utf8mb4
            COLLATE=utf8mb4_unicode_ci ";
        $this->adapter->exec( $sql );
    }

    public function updateRecord($table, $updateValues, $id = NULL) {
        $flagNeedsReturnID = (!$id);
        if ($flagNeedsReturnID) R::exec('SET @uuid = uuid() ');
        $id = parent::updateRecord( $table, $updateValues, $id );
        if ($flagNeedsReturnID ) $id = R::getCell('SELECT @uuid');
        return $id;
    }

    public function getTypeForID(){
        return self::C_DATATYPE_SPECIAL_UUID;
    }
}
```

Now you need to rewire the objects to swap the old Query Writer for the new one:

```
        $oldToolBox = R::getToolBox();
        $oldAdapter = $oldToolBox->getDatabaseAdapter();
        $uuidWriter = new UUIDWriterMySQL( $oldAdapter );
        $newRedBean = new OODB( $uuidWriter );
        $newToolBox = new ToolBox( $newRedBean, $oldAdapter, $uuidWriter );
        R::configureFacadeWithToolbox( $newToolBox );
```

Depending on the namespaces and aliases you use you might have to adjust this code accordingly.

Enabling UUID support in PostgreSQL

To install the UUID module for **PostgreSQL** run the following query:

```
    CREATE EXTENSION "uuid-ossp";
```

This command requires at least PostgreSQL version **9.1**, for earlier versions of Postgres please consult their documentation.

Here is an example class for PostgreSQL:

```
class UUIDWriterPostgres extends PostgreSQL {

    protected $defaultValue = 'uuid_generate_v4()';
    const C_DATATYPE_SPECIAL_UUID  = 97;

    public function __construct( Adapter $adapter ){
        parent::__construct( $adapter );
        $this->addDataType( self::C_DATATYPE_SPECIAL_UUID, 'uuid'  );
    }

    public function createTable( $table ){
        $table = $this->esc( $table );
        $this->adapter->exec( "
        CREATE TABLE $table (id uuid PRIMARY KEY); " );
    }

    public function getTypeForID(){
        return self::C_DATATYPE_SPECIAL_UUID;
    }
}
```

These are just examples, they allow you to use UUID but they may not fit your needs. I recommend to craft your own Query Writer, tailored to your needs.

Project

RB Adaptive

Since RedBeanPHP 4 the project has been split into two flavours. RedBeanPHP 4 Adaptive caters a different audience; those who like the core concepts of RedBeanPHP but rather see it like a toolbox or framework to build upon than a mere library. RedBeanPHP 4 Adaptive offers several features not found in the KS edition like:

- Integration/Installation with Composer
- Integration with PHPUnit
- Architecture Tooling

More details will follow...

Changelog

Welcome to RedBeanPHP 4.0, the new version of RedBeanPHP ORM. RedBeanPHP has been released on: 2014-04-01. RedBeanPHP 4.0 takes our little yet powerful ORM to the next level. This page discusses the general idea behind RedBeanPHP 4.0 and details the new features.

Release

I am very happy to announce the official release of RedBeanPHP 4.0, a brand new version of our ORM library. After 9 betas and 1.5 month of pro-active testing we are ready to deliver a robust and solid, future proof RedBeanPHP package. Some stats: 13177 unit tests (core only), 100% code coverage, only 2534 lines of 'effective' code (big clean up).

New in RedBeanPHP 4

RedBeanPHP 4 KS is a new version of the RedBeanPHP ORM library. The fourth version of RedBeanPHP is 'how I would have written RedBeanPHP' if I started right now instead of five years ago when the RedBeanPHP project started. I have removed lots of features. Some of them will reappear in plugins, others will not. The core of RedBeanPHP has been stripped of all experiments and legacy cruft build up over the years. RedBeanPHP 4 KS has a very clean, maintainable core incorporating many new concepts of the new PHP language: **namespaces** and **phar**. Some concepts have been simplified to be more in line with the 'on-the-fly' philosophy of RedBeanPHP, for instance, dependencies (to remove beans and add foreign keys) are no longer specified 'a priori', instead you add them on-the-fly by accessing the **x-own-list**. This new incarnation of RedBeanPHP should pave the way for future development of the library.

Features

This is a list of new features in vesion 4.0 of RedBeanPHP.

Namespaces

RedBeanPHP 4.0 is the first version to support native PHP namespaces, thus requiring PHP 5.3 or higher. While the code of RB has been namespaced properly, the R class is still available without a namespace, this has been done to avoid having to use the obnoxious 'use' statements all over the place. The R facade class and some other classes are loaded into global namespace by the phar-loader. This means that for most people nothing actually changes regarding the use of RedBeanPHP.

Phar distribution

RedBeanPHP was always about avoiding auto-loaders and configuring include paths. Therefore we always put the whole thing into one file. In a sense we were ahead of our time because this is of course exactly what the PHP PHAR functionality does. Phar allows us to support PHP namespaces while preserving the advantage of compiling everything into a single file for ease of use and ease of distribution.

Some IDEs have rather limited support for PHAR files. This is why we ship an ide-support.php file for your IDE containing the most important method definitions. Just keep this file around for your IDE to scan but do not include it. It should fix any autocomplete issues while IDE vendors are working to improve PHAR integration. Yes, I am looking at you *NetBeans*, *Eclipse*!

Big clean up

RedBeanPHP 4 is 'RedBeanPHP' as I would have written it today, instead of in 2009. This version is a major version and therefore it is **not backward compatible**. I took the liberty to remove all cruft accumulated over the years making the library slim and more maintainable. This new manual website covers only functionality found in version 4, the old version 3 manual will remain available and covers the other topics. Among things that have been removed are: Cooker (plugin and partially implemented in dispense), BeanCan Servers (now plugins), certain Query Writers (now plugins), Dependency Injectors (probably moves to RedBeanPHP Adaptive), old Association API and the Bean Cache.

CUBRID now a plugin

CUBRID Query Writer has been moved to the plugin repository. Instead of bundling all Query Writers in the core we will try to improve the plugin infrastructure for writers using additional (test)-hooks. This will help to keep the core slim and maintainable.

Exclusive own-list

The Exclusive own-list is a new feature in RedBeanPHP 4.0. This list replaces the R::dependencies method managing foreign keys. In short accessing an own-list with the x-prefix will cause the adding of a foreign key constraint with cascading deletes (only if used for the first time) and it will tell RedBeanPHP to trash all the beans in the list if the list gets emptied (breaking the association). Thus we 'translate' the concept of dependencies to the 'on-the-fly' principles of RedBeanPHP as it should be.

List-suffix

While RedBeanPHP manuals always warned against using plural bean type names (books) people did not like the resulting list names: ownPage, ownBook etc. Therefore instead of using **ownPage** you may now also open a list using: **ownPageList**. This also applies to shared lists.

Trees

The Traverse function is a new feature in RedBeanPHP 4.0. This method replaces the searchIn() method for searching trees.

Other features

Besides the big features mentioned above, there are also a lot of minor improvements. Here is a quick and incomplete list:

- 10% performance improvement for basic CRUD operations
- Performance improvements for bean conversion
- Improved Array Access interface (you can now use arrays instead of beans all the time)
- Improved handling of unique constraints
- Added EID() function to easily insert ENUM bean IDs in queries
- Constraints now also use ON UPDATE CASCADE
- Dispense works more consistently now
- Fixed an issue with type of return ID value in Postgres driver
- Fixed possible cache collision issue
- Performance improvements for fluid mode

Adaptive

As it turns out some people see RedBeanPHP more as a begin of something else. While RedBeanPHP 4 KS returns to the roots of RedBeanPHP, simplifying the entire library, the Adaptive branch will follow its own unique development path. Complex utilities like dependency injectors, query builders and pipelines will probably become available in the new RedBeanPHP Adaptive branch. So, for industrial strength RedBeanPHP visit our Adaptive friends !

Version 4 FAQ

This is a list of questions and answers regarding the 4.0 release.

What about PHP 5.3 - PHP 5.3.3 ?

These versions of PHP lack some important 5.3 features, however there is a special version in the download archive that supports these versions go to the archive and download RedBeanPHP 4 Beta 9b (same as FINAL 4.0 but with additional patch).

Why has the Preloader been removed?

When I wrote the preloader, the original purpose was to prevent for-each loops to fire queries when retrieving the parent of a bean. Later I added the writer cache which could take care of this but was turned off by default. In RedBeanPHP the writer cache is turned on by default solving the original problem. Meanwhile people requested all kinds of new features for the Preloader like support for loading own-lists, shared-lists and even aliases and SQL snippets. It even got its own syntax. I decided to remove the preloader because I believe simple SQL is better suited to query large amounts of records all at once for overviews and reports.
This functionality is still available as a plugin.

Why has graph() been removed from core?

R::graph() was a powerful feature to load and updates beans directly from forms. However the graph() function assumed you were also using FUSE for validation. Otherwise the function could lead to serious architectural and security defects. I fixed this in version 3, but then it became less powerful, so in version 4 I decided to remove it entirely from the core.
This functionality is still available as a plugin.
Also note that the new R::dispense() method works much like the old graph() method.

Why is R::associate gone?

The R::associate() method (as well as unassociate etc...) is a relic from the past. In the earliest versions of RedBeanPHP I believed you only needed many-to-many relations. Although this was true, performance became a real bottleneck. I had to find a way to apply the on-the-fly philosophy to N-1 relations as well, this resulted in the introduction of own-lists and shared-lists. Since then, I kept the old associate() method for backward compatibility reasons. In version 4 however I decided to finally clean up.

Why are the BeanCan Servers gone?

They blurred the distinction between plugin and core. Also, the RedBeanPHP Adaptive branch is going more in the direction of a framework which is a better place for BeanCan as well. RedBeanPHP 4 returns to the core of the library: on-the-fly ORM. Another reason is that it turns out it is pretty much impossible

to prescribe the interface of a JSON or REST API.
This functionality is still available as a plugin

Why does RedBeanPHP 4KS not support composer?

As of version 4 KS I revoke all support for Composer. There are several reasons for this:

- I don't want newcomers to learn about Composer first. The PHAR distribution is easy to use and does not require any prior knowledge about other systems
- I question the usefulness of systems like Composer/Gems/NPM - I think this should be handled manually
- Supporting Composer well, requires me to change the layout of my library (putting everything in a lib folder) which is ridiculous
- I don't want to make RedBeanPHP depending on Composer. Many libraries I use are completely messed up because their authors assume everyone is using Composer
- Packagist uses Github as a distribution tool - which is a mistake. Git is version control system and not meant to be used for this.

New in RedBeanPHP 3.5.7

This is a minor maintenance update.

- Fixed issue in QueryWriter cache (3.5.7b)
- Improved stacktrace in SQL exception
- Improved performance of convertToBeans() method
- Allow duplication of trees
- With now also works with joins

Roadmap

RedBeanPHP is actively developed by a community of open source developers.

Future versions

This roadmap can change every moment but my plan for RedBeanPHP is pretty simple. As of version 4.0 RedBeanPHP has reached full maturity. Therefore I do not intend to add many more features because that would only make the library bloated, instead I will focus on maintenance, fixing bugs (if there are any ;)), improving performance, testing and maybe add some convenience methods here and there.

RedBeanPHP 4.1 (October 2014)

- Improved support for UUIDs/GUIDs. This feature has been backported to *4.0.5* as well.
- Column functions: bind a function to a column
- Improve setup time by providing direct PDO setter (use with care!)
- Add a method to test the database connection
- Add new debug mode with query parameters filled in
- Add new debug function to inspect beans and arrays of beans
- Treat beans in own-list as shared list: aggregated list
- Regular maintenance & clean up
- Additional tests

Release cycle

The release cycle of RedBeanPHP is two times a year; a spring release and an autumn release. This means every six months there will be a new version of RedBeanPHP.

- Spring Beta release: **March**
- Spring Final release: **April**
- Autumn Beta release: **September**
- Autumn Final release: **October**

Versioning

RedBeanPHP uses a very sane version numbering system. The version number tells you something about the version; it has meaning. All RedBeanPHP versions have a version number. The version number consists of three parts; major, minor and point release.

```
Version X.X.X
```

Meaning:

```
Version MAJOR.MINOR.POINT
```

Major version

When the major version number increases, this means the new version is **NOT** backward compatible with all previous versions. Most of the time this means you better not use it in your current project if you are already using RedBeanPHP or you might have to make some changes to the project to make it work with the new version of RedBeanPHP. This is not always as bad as it sounds. For instance version 3 is not backward compatible with version 2, but only if you use the optimizers (which by default are turned off). So while this is a major version bump it's actually not that bad. However, while difference between 2 and 3 is relatively small, the gap between 1 and 2 was a really big one. Anyway whenever the major version number changes make sure you check the changelog to determine whether you can upgrade or not.

Minor version

A minor version change means new **features**! Minor versions don't break backward compatibiltity, they just mean new features have been added. Often, this goes hand in hand with changes in documentation or **bugfixes**. Therefore it's relatively **safe** to do a minor upgrade. Be sure though to check the changelog on the website. You might be able to take advantage of the new features!

Point version

A point version or point release happens when the last digit has been increased. Note that although you might assume a digit normally varies from 0-9, you might encounter minor and point releases like X.X.12 or X.30.X. Not sure if this will happen, however as RedBeanPHP matures you will see less major upgrades and more minor upgrades and point releases. A point release version is normally a maintenance version. This may include bugfixes, new tests, documentation changes or just some code cleanup. While it's always a good idea to scan the changelog most of the time you can be pretty sure there are no compatibility issues nor interesting new feature. Of course if you have reported an issue the point release can be quite interesting because the bug might have been fixed. In this case, the Github bug report number and the fix will be mentioned in the changelog.

About

RedBeanPHP is a simple, easy-to-use, on-the-fly object mapper, especially suited for RAD, prototyping and people with deadlines. RedBeanPHP creates tables, columns, constraints and indexes automatically so you don't have to switch between your database client (phpMyAdmin) and your editor all the time (this does not mean you will never have to use phpMyAdmin or SQL though, read on...). Also you don't have to write configuration files because RedBeanPHP simply infers the database schema from naming conventions. Because

RedBeanPHP saves a lot of time you can spend more time developing the rest of the application.

No Configuration

Most ORMs use configuration files (XML, INI or YAML) or some sort of annotation system to define mappings. These systems force you to map records to objects upfront. RedBeanPHP is different. Instead of using configuration it uses conventions; a very small set of rules. RedBeanPHP uses these conventions to infer relationships and to automate mappings. RedBeanPHP also helps you to follow these conventions by automatically building the initial tables and columns for you - which also saves a lot of time. This means there is no configuration, less boilerplate code and more time left to focus on the business logic, testing and documentation, thus boosting development productivity and code quality.

A bridge between objects and records

SQL is a powerful query language for relational databases. Most ORMs act like a wall, hiding SQL from you. RedBeanPHP on the other hand tries to integrate both technologies, thus acting more like a bridge. For instance, RedBeanPHP allows you to embed SQL snippets in ORM methods to tune the retrieval of related beans from the database. RedBeanPHP seeks to strike a balance between object oriented programming and relational database querying.

Code Quality

RedBeanPHP has been carefully architected to be concise and maintainable. The core codebase is tested daily using about 20.000 unit tests (100% test coverage) on local servers and a Travis CI environment. The codebase contains a lot of inline documentation, is fully object oriented and improves security by promoting PDO based prepared statements and parameter binding.

FAQ

Why do you use so much static functions? What about coupling?

That's only the Facade. Behind the facade you will find a landscape of elegant classes, see the API for advanced usage/more information. The **API** closely resembles the interface of the facade class.

Is it wrong to use the static facade functions?

If you're not planning to swap frameworks regularly you can rely on the easy-to-use static facade functions like **R::dispense()** and **R::load()** etc. People often complain about static methods but in reality many of those so-called pure **OOP** style projects tend to become heaps of powerless miniature objects and countless wirings. I don't believe that works very well.

Why does RedBeanPHP not protect me from race conditions?

Because I believe the best way to prevent race conditions is to use database **transactions**. RedBeanPHP offers simple functions to use transactions: R::begin(), R::commit() and R::rollback(). All you need to do is bundle your related queries together in a transaction by wrapping them in a begin-commit block. Not familiar with transactions yet? Read about Transactions on Wikipedia or read this discussion on StackOverflow.

Why is RedBeanPHP one file? Isn't that bad practice?

RedBeanPHP is distributed as one file to ease installation and deployment. The build script called **Replica** compiles the RedBeanPHP class files to one file. So in reality, RedBeanPHP is not one file, read more about Replica.

How active is RedBeanPHP?

RedBeanPHP is being developed quite actively by me and the RedBeanPHP community.

Why don't you implement my feature request?

Depends. RedBeanPHP is being developed in a very careful way. I try to keep RedBeanPHP clean yet comfortable. It's tempting to implement lots of features but that would make RedBeanPHP bloated. Feel free to write your own plugin or fork the project.

Why does RedBeanPHP not support custom table mapping (anymore)?

The idea of RedBeanPHP is to generate a useable and queryable schema based on your code and without any configuration. Custom table mappings don't fit very well in this model. However there are other reasons as well. Many so called power features like deep-copy have to make assumptions about database layout and table naming conventions. They can of course use some kind of configuration file to figure things out, but hey the whole idea of RedBeanPHP was **NOT** to use configuration!

In the past RedBeanPHP had a bean formatter for custom mappings, this functionality does not exist anymore. If you still require custom mappings, for instance to use RedBeanPHP with existing schemas you might want to try to use **VIEWS**. Simply map the views to your tables. If you only change table names and column names your views can be used for updates as well. Although not a perfect solution we have received some positive feedback about this approach.

Why does RedBeanPHP not provide a portable query language?

I do not believe in portable query languages or database independent query builders. The whole point of selecting a database is to choose the system that provides the most useful features. A portable query language by definition can't use database specific features, so you simply get the worst of all. Just dare to choose your the database system that fits the best for the task at hand.

Why are underscores and uppercase chars not allowed in type and property names?

Underscores ARE allowed in property names, just not in type names. RedBeanPHP uses underscores to denote relationships among beans. Uppercase characters cause problems on different operating system platforms. These characters have one further disadvantage; because programmers like me are often lazy, they get overused to form ambiguous words. The English vocabulary is quite big and you should better be creative and find the best word for the concept your bean or model describes. For instance; instead of "user_project" or "ProjectUsr" you can use "participant". This makes your database prettier and easier to read as well.

Checklist

Is your project suitable for use with RedBeanPHP ? It depends. Most of the time this is a personal choice. Personally I would use the following checklist to determine whether RedBeanPHP can be used for a certain project.

Suitable Projects

- Prototypes
- Import or conversion scripts
- Blog/Website CMS, e-commerce platforms, forums
- Small/medium sized business applications from scratch, less than 50 tables

- Phasing out old legacy applications with horrible schemas (remap using views and make the legacy code fun again!)
- Low power, embedded apps (with SQLite)

Less Suitable Projects

- Existing applications with custom database (use Propel instead)
- Hi-traffic distributed content apps, i.e. the next Twitter... (use a NoSQL database like Cassandra)
- Project requiring serious use of **UUIDs/GUIDs**. While RedBeanPHP offers basic **read** support for UUIDs, this is probably not sufficient.

You should also **NOT** use RedBeanPHP if you don't like the **RedBeanPHP schema policies** and you want complete control over the layout of your database schema, i.e. the column names used for primary keys and foreign keys. In this case I recommend to use: Doctrine. If Doctrine is too big for your taste you might also consider a small, **active record** like ORM written by a friend of mine: DicaORM.

Plugins

Here is a list of 3rd party plugins for RedBeanPHP, enjoy! Creating your own plugin for RedBeanPHP is easy! Did you create a plugin for RedBeanPHP? Send me an e-mail and I'll probably add it to the list!

ReBean

Plugin: ReBean, automatic revision management for RedBeanPHP.
Author: Zewa666

ReBean adds revision tables to your database and uses triggers to automatically insert revision beans.

StdErr Logger

Plugin:StdErr Logger, Logger that writes to StdErr.
Author: Zewa666

Logs queries to error log.

MySQL Backup Plugin

Plugin:MySQL Backup, Table exporter for MySQL
Author: Zewa666

Backups all tables in a MySQL database to a file. Only works for MySQL.

German Porter Stemmer Plugin

Plugin:German Porter Stemmer Plugin, a tool to improve search results in German language.
Author: Zewa666

A linguistic extension to improve search results for German language.

misc

Plugins:Create Your Own

RedBeanPHP / Plugins / **Create Your Own**

Plugins are an elegant way to add new functionality to RedBeanPHP. To create a plugin, create a class or function and use:

```
R::ext( 'doSomething', function() {
    return MyClass::myMethod();
} );
```

to add your new feature to the R-class (required PHP 5.3+ and RedBeanPHP 3.5+). For older versions of PHP use:

```
R::ext( 'doSomething',
    array( 'MyClass', 'MyStaticMethod' )
);
```

Now you can use your plugin like this:

```
R::doSomething();
```

it's that easy! Here is a list of some interesting 3rd party plugins for RedBeanPHP !

Installing a Plugin

Installing plugins is really easy. Most plugins, the legacy plugins for instance, automatically register their plugin functions with the R-facade for your convenience. So to install the Cooker plugin (one of the legacy plugins) use:

```
    require 'plugins/Cooker/Cooker.php';
```

Yes, that's all. Just include the file and you're done. Sometimes the plugin author has additional instructions and might require manual registration with the R-facade, this is mainly because the author is an object 'purist'. To install these plugins correctly consult the corresponding readme files or manuals.

Testing your plugin

You can use the RedBeanPHP unit test facilities to test your plugin. Simply write a test class extending RedUNIT\Base like this:

```
    class TestMe extends \RedUNIT\Base {
        public function testMethod()
        {
            ...
        }
    }
```

The RedUNIT test facility will simply run all methods starting with 'test'. The tests will run for every driver. If you want your test to run for a specific driver only extend RedUNIT\MySQL or RedUNIT\Postgres etc instead. To perform a test use the asrt($a, $b) command. If $a === $b the test will pass and the number of the test will be printed on the screen, otherwise the program will exit immediately printing an error message. A list of test commands:

```
    asrt( $a, $b ); //pass if a === b otherwise fail
    pass(); //count as passed
    fail(); //fail (i.e. die with error message)
    testpack( $message ); //print message
```

To run your tests:

```
php runtests.php "testing/Mytest.php" "RedUNIT\Base\Mytest"
```

First argument is the file to run, second argument is the name of the testing class contained in the file. You can not test more than one test file per plugin. Test coverage of your plugin will be calculated and printed. A report containing all missed lines will be saved to the cli folder. You can specify a third parameter to only count code lines of specific files, only lines in matching filenames will be counted in the code coverage statistics.

Writing a new Query Writer

To add a new, custom Query Writer for your favourite RDBMS simply wrap the writer in a plugin. To activate your writer people should issue the following command:

```
    //for instance to install DB2 database support
    require 'db2.php';
    R::setupDB2($dsn, $user, $pass);
```

The reason for this is that it is too hard for me to maintain all writers in the core, especially writers for commercial database platform I can't even obtain. Therefore they are separated from the core and hosted in their own repositories by the contributors. Query Writer authors can make the plugins available using the setup-approach described above. Note that this is also more flexible as it allows a database writer plugin to select a different driver (OCI instead of PDO for instance) or maybe even a different toolbox. Query writers should implement all methods defined in the Query Writer interface (see API). Many methods are already implemented in the Abstract Query Writer, these can be reused if they are appropriate for the new Query Writer as well.

Legacy Plugins

RedBeanPHP 4 KS goes back to the roots of RedBeanPHP, on-the-fly ORM. As such many additional functionalities and modules have been removed from this core. However they have been moved to a special plugin repository on github.

Replica

RedBeanPHP / Download / **Replica**

Replica is the build tool for RedBeanPHP. You can use Replica to build a all-in-one package yourself.

Run Replica2

To run replica2:

```
php replica2.php
```

Replica will now produce the following files:

```
rb.php
rb.phar
```

Now include the file of your choice in your PHP script.

Internals

PDO types

RedBeanPHP is a weakly typed ORM. It accepts all kinds of types in beans; integers, strings, booleans and NULL values. After a bean has been retrieved from the **database** each property of the bean contains a value of one of the following types: string, NULL, array or RedBean_OODBBean (object). RedBeanPHP will never return long values, booleans or integers. In fact, most values are returned as a string, with the exception of NULL which remains NULL. Composite types are also preserved and are limited to arrays and RedBean_OODBBean objects (embedded beans).

Value conversion in PDO binding

RedBeanPHP tries to convert data types by itself to preserve information. It's very important that you understand how RedBeanPHP deals with data types. If a value is numeric, the value will be bound to a prepared statement as an integer. However this is only the case if the integer representation is the same as a string representation. So while RedBeanPHP will bind 1900 as an integer, it will bind 007 as a string to preserve the padding zeros. Null values will be bound to statements using the NULL type. Also be careful with fractions. RedBean stores floats and doubles as doubles (bound as string). If you dont want this (to enable a higher level of data precision) I recommend to bypass RedBeanPHP and store these values yourself. Also consider using a proper Math library if working with high precision calculations.

Note that we talk here about PDO bindings, to set 007 in a bean property and preserve the zeros set the meta property:
$agent->setMeta("cast.agentname","string"); -- where agentname is the property and $agent is the bean.

Objects behind the Facade

Some people prefer to use objects instead of static methods. This is easy to accomplish, however it can be difficult to switch from one approach to another during a project, so I recommend to give this some thought. Personally I always rely on the facade and its static methods, even in big projects. This is because I favour simplicity over pure object oriented code, but this is personal preference of course. Learn how to use RedBeanPHP without the static facade.

Credits

RedBeanPHP has been written by Gabor de Mooij and the RedBeanPHP community. I would like to thank all of the contributors for their effort. Without your help it would not have been possible to maintain this wonderful project.

Thanks to:

daviddeutsch
tomasklapka
damianb
seanhess
murich
jstsch
hugollm
SteveEdson
brianhaveri
agvstin
F21
gaving
zerotri
midnightmonster
palicao
daandavidsz
kadishmal
etisfo
saetia
michaelklishin
m6w6
marcioAlmada
rlerdorf
sandulungu
zebulon303
luniki

Without your help, RedBeanPHP would not be such a great project. Thank you! Let's make RedBeanPHP even better!

Special thanks to Zurmo (for using RedBeanPHP and promoting it), Sean Hess (for using and promoting RedBeanPHP when it did not even have its own site), Wouter Toering (for the CSS), Erik Roelofs (for the inspiration to write RedBeanPHP), David Deutsch (for support and contributions), Zewa (for your plugins), Richard Keizer (for support), Robin Mogre (for helping me with the design in version 2) and Robert Cabri (for support).

Did I forget to mention you ? Please don't feel offended. Just drop me a mail, sometimes it's hard to track all of you, especially because we are such a dynamic community.

License

RedBeanPHP has been dual licensed New BSD and GPLv2.

New BSD License

Disclaimer

GPL v2 License

Version 2, June 1991

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would

not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this.

Archives

Old manuals can be found here:

- Manual Version 3
- Previous Additional Manual 3.3 and earlier
- Manual RB 2.0

Download Archive

Searching for a specific version of RedBeanPHP from the past? Consult the download archive page for all previous RedBeanPHP products.