Getting started

System Requirements

RedBeanPHP has been tested with PHP 5.3 but runs under PHP 5.2 as well. RedBeanPHP works on Mac OSX, Linux and Windows. You need to have PDO installed and you need a PDO driver for the database you want to connect with. Most PHP stacks and distributions come with PDO and a bunch of drivers so this should not be a problem.

Installing

To install **RedBeanPHP**, download the RedBeanPHP All-in-one pack from the website. After unzipping you will see just one file:

```
rb.php
```

This file contains everything you need to start RedBeanPHP. Just include it in your PHP script like this:

```
require('rb.php');
```

You are now ready to use RedBeanPHP!

Setup

So, you have decided to start with **RedBeanPHP**. The first thing you need to get started is setting up the **database**. Luckily this is really easy.

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

Yes, that's all if you are working on a UNIX, Linux or Mac system with **SQLite**. Here is how to connect to **MySQL** on any machine:

```
require('rb.php');
R::setup('mysql:host=localhost;dbname=mydatabase','user','password');
```

RedBeanPHP is also very easy to setup for use with PostgreSQL and SQLite.

RedBeanPHP only works with the InnoDB driver for MySQL. MyISAM is too limited.

Compatible

RedBean PHP has fluid and frozen mode support for:

MySQL 5 and higher
SQLite 3 and higher
PostgreSQL 8 and higher

To connect with these **databases** use:

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

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

R::setup('sqlite:/tmp/dbfile.txt','user','password'); //sqlite
```

Basics

Create a Bean

Using RedBeanPHP is easy. First create a bean. A bean is an object with public properties. Every bean has an id and a type.

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

Now we have an empty bean of type book. Now, add properties:

```
$book->title = 'Boost development with RedBeanPHP';
$book->author = 'Charles Xavier';
```

Now store it in the database:

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

That's all? Yes. Everything has been written to the **database** now! RedBeanPHP automatically creates the required table, columns and indexes. Use the load() function to retrieve the bean from the database again.

How does it work?

Our bean has two properties, both strings. Thus RedBeanPHP will create a VARCHAR(255) column in the database (in this case MySQL). If we add a property $book->price = 100, RedBeanPHP will add a TINYINT column. If we later add $book->price = 99.99 RedBeanPHP will alter the column to make it possible to store a floating point number. If you pass an SQL date like 2012-01-01 or a date-time value like 2012-01-01 12:00:00 RedBeanPHP will add a date/datetime column, if a column already exists a suitable type will be used that does not affect other data. You can also use MySQL spatial data types i.e. $bean->place="POINT(2 1)". A Spatial value string will be converted to its binary representation.

Note that the store() function returns the ID of the record.

While $bean->id often returns the ID of a bean, a safer way to access the ID of a bean is to use $bean->getID(); (this will return the ID even if it has been stored in a different property)

RedBeanPHP only supports lowercase bean types and lowercase properties. While RedBeanPHP won"t break in case of uppercase property or type names, you might hit some compatibility issues later on. Also note that case sensitivity in not supported by some databases and operating systems.

Learn how to connect to a database with a single command.

Loading a Bean

To load a bean from the database use the load() function and pass both the type of the bean and its id:

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

This code loads the previously stored bean of type 'book' from the database. If the bean cannot be loaded a new empty bean will be dispensed with id 0. To check whether a bean has been loaded correctly you can thus verify the id using

```
if (!$bean->id) { ...help bean not found!!.. }
```

To access data in the bean, simply access the public properties like this:

```
echo $book->title;
```

Besides loading a single bean, you can load a batch of beans at once:

```
$books = R::batch('book',array($id1,$id2));
```

To count beans:

```
R::count($type);
```

Finally, you need to be able to remove beans...

If the bean cannot be loaded a new empty bean will be dispensed.

Deleting a Bean

To remove a bean from the **database**:

```
R::trash( $book );
```

To delete all beans of type book:

```
R::wipe( 'book' );
```

For a complete overview of all methods in R, see the facade API.

Freeze

By default **RedBeanPHP** operates in fluid mode. In Fluid mode the **database schema** gets changed while you are working to meet the requirements of your code. This is ideal for development. During development you can just make beans, put them in the database and don't worry about the rest. When you are done developing you should review the database schema and make manual adjustements where necessary. Then, you can freeze RedBeanPHP with the freeze() command. This will lock the schema to prevent further modifications.

```
R::freeze( true ); //will freeze redbeanphp
```

Things to do after freezing the schema:

1. Check the column types
2. Configure the foreign keys*
3. Add additional indexes for better performance

*RedBeanPHP will add foreign key constraints for N:1 relations. For instance if a page belongs to a book and you throw away the book, the page will remain in the database and the reference to the book record will be set to NULL. If you want to make the page dependent on the book you should change this to CASCADE in phpmyadmin. Then the page will be thrown away once the book has been deleted. It's impossible for RedBeanPHP to determine the the desired action upfront so you have to configure this afterwards.

Don't forget to freeze and review the **database** schema afterwards!

Finding

Finding Beans

RedBeanPHP allows you to use good old SQL to find beans:

```
$needles = R::find('needle',' haystack = ?', array( $haystack ));
```

Find takes three arguments; the type of bean you are looking for, the SQL query and an array containing the values to be used for the question marks. You can also use named slots in your queries:

```
$needles = R::find('needle',' haystack = :haystack
                              ORDER BY :sortorder',
array( 'sortorder'=>$sortorder, ':haystack'=>$haystack ));
```

To find all needles:

```
$all = R::find('needle');
```

To find all needles without conditions but with **ORDER** or **LIMIT**:

```
$all = R::find('needle',' 1 ORDER BY title LIMIT 2 ');
```

The SQL snippet starts with 1, this is because the find() functions expect a WHERE condition. Instead of an array, find() can return just the first record:

```
$all = R::findOne('needle',' 1 ORDER BY title LIMIT 1 ');
```

You can use IN-clauses as well:

```
$all = R::find('needle',' id IN ('.R::genSlots($ids).') ',$ids);
```

If no beans are found, find() returns an empty array and findOne() returns NULL.

the 'WHERE' keyword is not necessary in find()

Queries

If you prefer rows rather than beans you need to execute an SQL query, you can use the query functions of **RedBeanPHP**. To just 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 multi dimensional array:

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

    [1] => Array
        (
            [id] => 2
            [title] => back
            [text] => bye
        )
)
```

Note that you can use bindings here as well:

```
R::getAll( 'select * from page where
title = :title',
array(':title'=>'home') );
```

To fetch a single row:

```
R::getRow('select * from page where title like ? limit 1',array('%Jazz%'));
```

The resulting array corresponds to one single row in the result set. For instance:

```
Array
(

    [id] => 1
    [title] => The Jazz Club
    [text] => hello
)
```

To fetch a single column:

```
R::getCol('select title from page');
```

The result array of this query will look like this:

```
Array
(

    [0] => frontpage
    [1] => The Jazz Club
    [2] => back

)
```

And finally, a single cell...

```
R::getCell('select title from page limit 1');
```

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

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

result:

```
Array
(

    [1] => frontpage
    [2] => The Jazz Club
    [3] => back
```

```
)
```

It's easy as that!

Mixing SQL and PHP

In RedBeanPHP you can mix PHP and SQL as if it were just one language. To call an SQL function in PHP simply call it like a PHP function on R::$f. $f is short for 'function' and we mean SQL function here. Here are some examples:

```
$time = R::$f->now(); //executes and returns result of: SELECT NOW();
```

Besides simple SQL functions like now() you can construct queries that blend perfectly with your PHP code. This is useful for writing complex dynamic queries. Also this technique allows you to share queries among functions and methods just like a query builder does. The only difference is, you don't need to learn a new syntax, it's plain old SQL!

```
R::$f->begin()
->select('*')->from('bean')
->where(' field1 = ? ')->put('a')->get('row');
```

This PHP code is the equivalent of:

```
select * from bean where field1 = 'a'
```

There are just a couple of rules. First, you must begin a php-sql query that is longer than just one SQL function with the begin() method. This method prepares the SQL helper for a query. Now you can chain SQL statements as PHP function calls. To add a value to the parameter list use put(), finally you must end the query with get(), get('row'), get('col') or get('cell'). These methods are similar to the default database adapter methods found elsewhere in RedBeanPHP. Thus get() will return a multi dimensional array with each row containing an associative array (column=>value), while get('row') returns just one such row. Analogous get('col') returns a flat array of the column values and get('cell') returns a single value.

Relation Mapping

Adding Lists

A bean can contain other beans. To add beans to a bean you add a **list**. A list is actually nothing more than just a property that contains an **array**. There are 2 kinds of lists, the own-list and the shared-list. To add a list to a bean add a property own**X** or shared**X** where X is the type of beans that are meant to be in the list. If a list has been filled once, the old beans are loaded in the list the moment you access it (we call this **lazy loading**). So to retrieve all the pages of a book:

```
$pages = $book->ownPage;
```

The own-list contains beans exclusively owned by that bean, the shared list contains beans that can be owned by multiple beans. Let's focus on the own-list. If you put a bean in one own-list and then move it to another, the other bean steals the bean from the former. Now let's see how to use own-lists. A village contains buildings; a building can only be in one village at a time:

```
list($mill,$tavern) = R::dispense('building',2);
$village->ownBuilding = array($mill,$tavern); //replaces entire list
R::store($village);
```

The exampe creates two buildings; a mill and a tavern and puts them in the ownBuilding list of the village. They now belong to the village. If there were any buildings already in this list they have now been overwritten because of the assignment. Use $village->ownBuilding[] to add them instead of replacing the entire list. Storing the village will also store all the added beans (if needed) and store the relationships. If you store the village and reload it, you will find your buildings in the property ownBuilding. Note that the name of the property has to match the type of beans you are storing in it, ownBuilding can only contain buildings, this is why the list is called ownBuilding. If you store pages, use ownPage etc. In the database, the buildings will be stored in a table called 'building', because that is the type of the bean. Each building will have a link field to the village named village_id that references the appropriate record in the village table.

To add a building later on, without removing older buildings:

```
$house = R::dispense('building');
$village->ownBuilding[] = $house;
```

To delete a building from the list use unset()...

```
unset($village->ownBuilding[2]); //2 is the id of the building, keys are ids
```

```
R::store($village);
```

Of course you can also change buildings in the list:

```
$village->ownBuilding[1]->name = 'The Old Inn';
```

To access a bean in a list, use the ID of the bean as a key,
i.e. $village->ownBuilding[$building_id] = ...

Bean lists are unsorted. Use usort() to sort beans.

Shared Lists

A building can only be in one village; so in the previous example we used ownBuilding[] to store buildings. In other situations however you want to share objects. In this case use a shared list:

```
$army = R::dispense('army');
$village->sharedArmy[] = $army;
$village2->sharedArmy[] = $army;
```

Now both villages have the same army. Once again the name of the shared list property needs to match the type of bean it stores. In the database, RedBeanPHP will make a link table army_village to associate the armies with their villages. This is why it's called a shared list, because it's possible for one village to have the same army as another village. Contrast this to the table structure for villages and buildings as discussed in the previous chapter.

Nested Bean

You have seen in the previous chapters how to add a building to a village. But how about getting the village a building belongs to? To add buildings to a village we used something like $village->ownBuilding[] = $building. If we now start from the building and we want to obtain a reference to the village this building belongs to:

```
$village = $building->village;
```

You can also change the parent object if you like however I consider this bad practice. To break the relation with the parent object use:

```
unset($building->village); //works, neat everyone knows what you mean
$building->village = null; //works, bit messy
$building->village = false; //ugly
```

Trying to assign a something other than a bean to a parent object field will throw an exception:

```
$building->village = 'Lutjebroek'; //throws a RedBean Security exception
```

This means that once a property has been used to store a bean, it can only be used to store a bean afterwards.

Note that the name of the property needs to match the type of bean it stores.

Aliasing

Normally a property that contains a bean needs to be named after it's type. We have seen this with parent objects; to access the village a building belongs to you just read the $building->village property. 90% of the time this is exactly what you need. A parent object can be aliased though which is useful in more complex relations.

When dealing with people you often have to store persons using a role name. For instance, two people are working on a science project. Both people are in fact 'person' beans. However one of them is a teacher and the other is a student. The person thus has ownProject, but two persons can share a project because they have a different role; i.e. teacher and student.

```
list($teacher,$student) = R::dispense('person',2);
$project->student = $student;
$project->teacher = $teacher;
```

RedBeanPHP will store both the student and the teacher as persons because RedBeanPHP simply ignores the property name when saving. The columns teacher_id and student_id will be used just as you expect. However when you retrieve the project from the **database**, you need to to tell RedBeanPHP that a student or teacher is in fact a person. To do so, you have to use the fetchAs() function:

```
$teacher = $project->fetchAs('person')->teacher;
```

You can also built your own way to resolve types. See Advanced Aliasing.

You cannot use an Alias with R::view().

Trees

By nesting beans in other beans you can easily make trees. In the following example we are decorating a christmas tree with candy canes.

```
$canes = R::dispense('cane',10);
$i = 0;
foreach($canes as $k=>$cane) $canes[$k]->label = 'Cane No. '.($i++);
$canes[0]->cane = $canes[1];
$canes[1]->cane = $canes[4];
$canes[9]->cane = $canes[4];
$canes[6]->cane = $canes[4];
$canes[4]->cane = $canes[7];
$canes[8]->cane = $canes[7];

$id = R::store($canes[0]);
$cane = R::load('cane',$id);
asrt($cane->label,'Cane No. 0');
asrt($cane->cane->label,'Cane No. 1');
asrt($cane->cane->cane->label,'Cane No. 4');
asrt($cane->cane->cane->cane->label,'Cane No. 7');
asrt($cane->cane->cane->cane->cane,NULL);
```

All asrt() functions will return TRUE meaning the assertion is correct.

Models

Models and FUSE

A model is a place to put validation and business logic. Although you can put validations in your controller that would require you to copy-and-paste it whenever you need it. So putting validation and business logic into a central place saves you a lot of work. Models are connected to beans using **FUSE**; the best thing about this that everything happens automatically; there is no wiring to be done.

Imagine a jazz band that has place for 4 members, in this case we need to add a validation rule 'no more than 4 band members'. We could add this rule to the controller:

```
    if (count($_post['bandmembers'] > 4) ...
```

But like I said, then we need to copy this code to every controller that deals with band members. Now let's define a band model to see how this works with FUSE:

```
class Model_Band extends RedBean_SimpleModel {
```

```
    public function update() {
        if (count($this->ownBandmember)>4) {
            throw new Exception('too many!');
        }
    }
}
```

This model contains an update() method. FUSE makes sure that the update() method will get invoked as soon as we try to store the bean:

```
$band = R::dispense('band');
$musicians = R::dispense('bandmember',5);
$band->ownBandmember = $musicians;
R::store($band);
```

This code will trigger an exception because it tries to add to many band members to the band model. As you can see, the model is automatically connected to the bean; we store the bean using R::store() and update() is called on a populated instance of Model_Band. Just like update there are several other hooks:

Action on bean	**Invokes method on Model**
R::store	$model->update($bean)
R::store	$model->after_update($bean)
R::load	$model->open($afterOpenedBean)
R::trash	$model->delete($beforeDeletedBean)
R::trash	$model->after_delete($theOldBean)
R::dispense	$model->dispense($freshBean)

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;
```

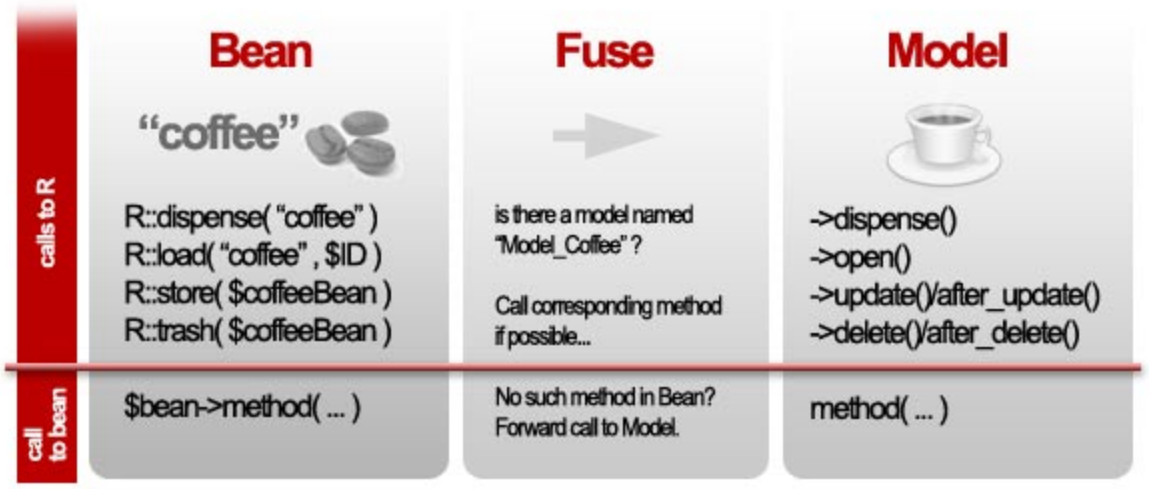
output:

```
 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"}
```

In the model, $this is bound to the bean. As is $this->bean.

How Fuse Works

Fuse adds an event listener (Observers) to the RedBean object database. If an event occurs it creates an instance of the model that belongs to the bean. It looks for a class with the name Model_X where X is the type of the bean. If such a model exists, it creates an instance of that model and calls loadBean(), passing the bean. This will copy the bean to the internal bean property of the model (defined by the superclass [SimpleModel]). All bean properties will become accessible to $this because Fuse relies on magic getters and setters.



Remapping models

By default RedBeanPHP maps a bean to a model using the Model_X convention where X gets replaced by the type of the bean. You can also provide your own mapper, here is how:

```
RedBean_ModelHelper::setModelFormatter( new MyModelFormatter );
```

Here we tell RedBeanPHP to use the MyModelFormatter class to find the correct bean-to-model mapping. This class looks like this:

```
class MyModelFormatter implements RedBean_IModelFormatter {
    public function formatModel($model) {
        return $model.'_Object';
    }
}
```

This class will make sure that a bean of type 'coffee' will be mapped to Coffee_Object instead of Model_Coffee.

In the model, $this is bound to the bean. As is $this->bean.

Fuse Custom Method

FUSE does not only support hooks like update() and delete(). You can also call a non-existant method on a bean and it will fire the corresponding method on the model.

```
class Model_Dog extends RedBean_SimpleModel {
        public function bark() {
                echo 'Whaf!';
        }
}

$dog = R::dispense('dog');
$dog->bark(); //echos 'Whaf!'
```

Forms and Cooker

The cooker is a tool to turn forms into beans. Instead of converting request arrays from forms (or JSON/XML) manually you can let the cooker take care of this.

```
<form>
    <input type="hidden" name="musician[type]" value="bandmember" />
    <input type="text" name="musician[name]" value="" />
    <input type="hidden" name="musician[ownInstrument][0][type]" value="instrument" />
    <input type="text" name="musician[ownInstrument][0][name]" value="" />
</form>
```

Take a look at this little pseudo form. If you fill this form with band member name: 'Joe' and instrument name: 'Kazoo', you can turn this into a bean structure:

```
$musicianBean = R::graph($_POST['musican']);
```

Your musician bean will now look like this:

```
Musician Bean
    name: Joe
    ownInstrument:
         Instrument Bean
             name: Kazoo
```

There are just a few basic rules. To identify an array as a bean it must contain a key called type that has the value of the bean type. Also you must stick to the naming conventions for beans and nested beans like ownBean and sharedBean.

By default the Cooker will also import empty beans, if you want the cooker to ignore beans without any properties except the id property or beans that only have empty properties you can use R::graph($array,TRUE). This is handy for forms that contain an additional input elements to add new items.

Demo App

For your convenience I have written a little demo application. Actually it's just a form. The real code is just two lines; one R::graph() and one R::store(). This shows the power of RedBeanPHP. The demo app is just a simple image based inventory system or mini database. Because I use it for my whisky collection I named it Malt Management System, however you can manage whatever you like with it and it's just one file of course.

Database

Schema

RedBeanPHP generates a sane and readable database schema for you. Prior to RedBeanPHP 3 this schema could be customized. However I have decided no longer to support this because the whole idea of RedBeanPHP is to favor convention over configuration. This is what makes it a powerful lib. Here are the schema rules for RedBeanPHP; the convention.

Table name:	Should match bean type
Primary key:	Each table should have a primary key named 'id' (int, auto-incr)
Foreign key:	Format: <TYPE>_id
Link table:	Format: <TYPE1>_<TYPE2> sorted alphabetically

Multiple databases

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

```
R::addDatabase('DB1','sqlite:/tmp/d1.sqlite','user','password',$frozen);
```

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

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

Transactions

RedBeanPHP offers three simple methods to use database transactions: R::begin(), R::commit() and R::rollback(). A transaction is a unit of work performed within a database management system (or similar system) against a database, and treated in a coherent and reliable way independent of other transactions. To begin a transaction use R::begin(), to commit all changes to the database use R::commit() and finally to rollback all pending changes and make sure the database is left untouched use R::rollback(). Usage:

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

The RedBeanPHP transactional system works exactly like conventional database transactions. Because RedBeanPHP throws exceptions, you can catch the exceptions thrown by methods like R::store(), R::delete(), R::associate() etc, and perform a rollback(). The rollback() will completely undo all the pending database changes.

If you are new to transactions, consider reading about database transactions first.

Many databases automatically commit after changing schemas, so make sure you test your transactions with after a R::freeze(true); !

Nuke

The R::nuke() command just does what you think it does. It empties the entire database. This is really useful for testing purposes. R::nuke() only works in fluid mode to prevent any damage. Usage:

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

Like other nuclear tools, nuke() should be used with care.

This feature is available since RedBeanPHP 2.2+

BeanCan

BeanCan Server

BeanCan is a php class that can act as a backend server for javascript centered web applications (JSON-RPC standard 2.0 compliant JSONRPC). In a JS based web application your views and controllers are written in client-side Javascript while your models are still stored on the server. BeanCan acts as bridge between the client side javascript views and controllers on the one hand and the server side models on the other.

BeanCan makes use of FUSE. This means that you can send 4 types of commands to the BeanCan Server:

load
store
trash
custom

Requests 1-3 are handles automatically by RedBean. This means you can store/delete/load any bean automatically if you connect to the bean server without any effort. If you send an unrecognized command, FUSE tries to locate the model and passes the request.

The following request returns a page with ID 1:

```
{
"jsonrpc":"2.0",
"method":"page:load",
"params":[1],
"id":"myrequestid"
}
```

The following request creates a new page and returns its new ID:

```
{
"jsonrpc":"2.0",
"method":"page:store",
"params":[{"body":"lorem ipsum"}],
"id":"myrequestid"
}
```

The following request changes the text of page 2:

```
{
"jsonrpc":"2.0",
"method":"page:store",
"params":[{"body":"welcome","id":2}],
"id":"myrequestid"
}
```

This example request deletes page with ID 3:

```
{
"jsonrpc":"2.0",
"method":"page:trash",
"params":[3],
"id":"myrequestid"
}
```

And finally this request executes $page->mayAccess($ip) and returns the result. FUSE will connect automatically to the Model_Page class to accomplish this.

```
{
"jsonrpc":"2.0",
"method":"page:mayAccess",
"params":[ ipAddress ],
"id":"myrequestid"
}
```

The BeanCan server returns JSON reponses like this (created page and returns ID):

```
{
"jsonrpc":"2.0",
"result":"8",
"id":"myrequestid"
}
```

In case of an error:

```
{
"jsonrpc":"2.0",
"error":{"code":"-32603","message":"Invalid request"},
"id":"myrequestid"
}
```

Full Example

Here is a full example.

```
require("rb.php");
R::setup();
class Model_Todo extends RedBean_SimpleModel {
public function getList() {
return R::findAndExport("todo");
}
}
$beancan = new RedBean_BeanCan;
if (isset($_POST["json"])) die($beancan->handleJSONRequest( $_POST["json"] ));

<html>
<head><title>DEMO BEANCAN</title>
<script src="jq.js"></script>
</head>
<body>
<h1>My Todo List</h1>
<ul class="list">
 foreach(R::find("todo") as $todo):
<li><button todo=" echo $todo->id; ">DONE</button> echo $todo->description; </li>
 endforeach;
</ul>
<fieldset>
<label>Todo:<label>
<input type="text" id="dscr" />
<button>add to list</button>
</fieldset>
<script>
$("document").ready(function(){
//Delete an Item
var requestDelete = '{"jsonrpc":"2.0","id":"delete","method":"todo:trash","params":[@]}';
var clickHandler = function(){
var $btn = $(this);
$.post("?",{"json":requestDelete.replace("@",$btn.attr("todo"))},function(d){
eval("data="+d);
if (data.id=="delete") $btn.parent().remove();
});
}
//Attach Click Handlers
$("li button").click(clickHandler);
//Add an Item
$("fieldset button").click(function(){
var requestAdd =  '{"jsonrpc":"2.0","id":"add_todo","method":"todo:store","params":
[{"description":"@"}]}';
$.post("?",{"json":requestAdd.replace("@",$("#dscr").val())},function(d){
eval(" data = "+d);
if (data.id=="add_todo") {
$(".list").append("<li><button todo='"+data.result+"'>DONE</button>"+$("#dscr").val()+"
</li>");
//Restore Click Handlers
$("li button").unbind().click(clickHandler);
}
```

```
});
});
});
</script>
</body>
</html>
```

Or download a sample package here:Sample BeanCan Server Demo

REST server

In RedBeanPHP 3.0 the BeanCan server also responds to RESTFul GET requests. To setup a REST server with beancan:

```
$server = new RedBean_BeanCan();
$server->handleRESTGetRequest('/book/2'); //returns book with ID 2
$server->handleRESTGetRequest('/book'); //returns books
```

Advanced

Custom getters

Besides, properties, nested beans and nested lists a bean may also contain custom getters. If you want to map a relation which is not supported by **RedBeanPHP** you can use this technique.

For instance, consider the retrieval of historical data. We have a $person bean and our task is to get all the related $events sorted by date. What RedBeanPHP can do is retrieve all events like this

```
$person->ownEvents
```

But that might not be the correct order! In this case we better write our own getter:

```
class Model_Person extends RedBean_SimpleModel {
    public function getEvents() {
        return R::find('event',' person_id = ? order by `date` asc',array($this->id));
    }
}
```

To use this custom getter:

```
$person = R::load('person',$certainID);
$eventBeans = $person->getEvents();
```

Notice that RedBeanPHP **knows** that Model_Person belongs to bean $person. This is called FUSE because the bean and the model are fused. Read more about models and fuse.

The flexibility of FUSE allows RedBeanPHP to be used as a stand-alone ORM solution, the BeanCan Server

Association API

Another way to use many-to-many relations is to use the R::associate() function. This function takes two beans and associates them. To get all beans related to a certain bean you can use its counterpart R::related()

```
R::associate( $book, $page );
R::related( $page, 'book' );
R::relatedOne( $page, 'book'); //just the first bean
```

To break the association between the two:

```
R::unassociate( $book, $page );
```

To unassociate all related beans:

```
R::clearRelations( $book, 'page' );
```

Sometimes you want to know which beans aren't associated with a certain bean:

```
R::unrelated( $scandal,  'politician', $sql, $values );
```

Are Related

To find out whether two specific beans are related, use the R::areRelated() function.

```
R::areRelated( $husband, $wife );
```

This function returns TRUE if the two beans have been associated using a many-to-many relation (associate) and FALSE otherwise.

Association and SQL

With the Association API it's possible to include some SQL in your relational query:

```
R::related( $album, 'track', ' length > ? ', array($seconds) );
```

Copy Beans

To copy a bean use the R::copy() function.

```
R::copy($book,'author');
```

The second argument should be a comma-separated list of many-to-many associations that need to be copied as well. Note that copy() won't copy any N:1 relations.

Copying Beans in 3.0

The copy() function is not very good. In RedBeanPHP 3.0 it has been replaced with a far more effective method called R::dup().

R::dup() makes a deep copy of a bean properly and without storing the bean. All own-beans will be copied as well. And all shared beans will be shared with the bean. Everything goes automatic. 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);
```

Complete Guide

Igor Couto has written an full guide on how to use beans to establish all sorts of database relations. I believe this guide is an excellent resource for everyone working with RedBeanPHP. Among other subjects this guide discusses simple relationships, one-to-many, many-to-many, one-to-one, self-referential, poly-one, poly-many etc. Feel free to contribute. Besides the PDF, there is also a pages edition and a MS Word edition.

Tags

Tagging means labelling stuff. Tags are often used to categorize or group items into meaningful groups. In RedBean tagging has been build into the facade. Tagging beans is very easy. For instance, imagine we have a bean of type "page" and we would like to attach several tags to this bean "topsecret" and "mi6". Now here is how we attach these labels to the page bean:

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

As you see, the tagging system accepts a comma-separated string which could come directly from your form module. However, the tag method also accepts arrays:

```
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 );
```

In this case the contents of variable $tags will equal the following string:

topscret,mi6

If you rather prefer an array you need to explode the return value of the function like this:

```
$tagItems = explode( ',', R::tag($page) );
```

To untag an item use

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

To get all beans that have been tagged with $tags:

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

To find out whether beans have been tagged with specific tags:

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

To add tags without removing the old ones (since RedBeanPHP 2.1):

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

Swap

It's very common in real life applications to swap properties. For instance, in a CMS you often have a feature to change the order of pages or menu items. To swap a property use:

```
$books = R::batch('book',array($id1,$id2));
R::swap($books,'rating');
```

We simply load two books using the batch loader, then we pass the array with two books to swap() as well as the name of the property we which to swap values of.

Import and Export

Websites often contain forms or have to process some sort of request array. Of course you can manually copy values from this array and put them in a bean like this:

```
$bean->title = $_POST['title'];
```

But that can become quite boring... so we offer a shortcut to load entire arrays directly into beans:

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

The code above is handy if your POST request array only contains book data. It will simply load all data in 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.

You can export the data inside a bean to an array, there are two functions to facilitate this:

```
$bookArray = $book->export();
```

Calling export() on a bean will export the data of a single bean into an array.

Recursive Export

To recursively export one or an array of beans use:

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

Import functions do not validate user input.

You can use models to add validation on-the-fly.

Debug

The **RedBeanPHP** debugger displays all queries on screen. Activate the debugger using the debug() function:

```
R::debug(true);
```

To turn the debugger off:

```
R::debug(false);
```

Meta Data

Beans contain meta information; for instance the type of the bean. This information is hidden in a meta information field. You can use simple accessors to get and modify this meta information.

To get a meta property value:

```
$value = $bean->getMeta('my.property', $defaultIfNotExists);
```

The default default value is NULL.

To set a meta property:

```
$bean->setMeta('foo', 'bar');
```

The type of the bean is stored in meta property 'type' and can be retrieved as follows:

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

Since 3.0: Meta data can be used for explicit casts. For instance if you want to store something as a POINT datatype:

```
    $bean->setMeta('cast.myproperty','point');
```

Unique

Usage of build command unique meta property:

```
$bean->setMeta("buildcommand.unique" , array(array($property1, $property2)));
```

Public Meta properties

Here is an overview of all public meta properties used by the system. These meta properties are safe to read and can be used reliably to extract information about beans. Don't change them though!

Property	Function
type	(string) Determines the type of the bean, don't change!
tainted	(boolean) Whether the bean has been modified.

Private Meta properties

Here is an overview of all system meta properties. These meta properties should not be relied on, they are only for RedBeanPHP internal subsystems.

Property	Function
buildcommand.unique	issues an extra option for query writer, use with care
buildcommand.indexes	issues extra options for query writer, for internal use only
cast.*	Used for explicit casting, for internal use only

Property	Function
buildreport.flags.*	Information about internal processes
sys.*	System information, never touch!
idfield	Holds the current ID-field, use getIDField() instead.

Tainted

Sometimes its useful to know whether a bean has been modified or not. The current state of the bean is stored in a Meta property called **tainted**. To get the state of the bean use:

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

If the bean has been modified this will return boolean TRUE, otherwise it will return FALSE.

Extended Associations

An extended association is a many-to-many association with a some extra information.

```
R::associate($track,$album,array('sequencenumber'=>$s));
```

JSON is also allowed

```
R::associate($track,$album,'{"order":"2"}');
```

Or just a string

```
R::associate($track,$album,'2'); //stored in property 'extra'
```

To load a association link:

```
$keys = R::$extAssocManager->related($album,'track');
```

Architecture

Architecture

RedBeanPHP has a beautiful, clean and easy to understand architecture. In essence, RedBeanPHP consists of three basic core classes: OODB, Adapter and QueryWriter. Each core class has its own task. The task of the Adapter is to connect to a **database** driver, the task of the QueryWriter is to write database specific SQL and execute it using the adapter. The task of OODB is to perform CRUD using the QueryWriter. Because RedBeanPHP supports many databases there are several versions of QueryWriter, all implementing the same QueryWriter interface. While RedBeanPHP only supports PDO, it's possible to write alternative Adapters implementing the same Adapter interface.

The toolbox is nothing more than a combination of OODB with a specific Adapter and QueryWriter. In essence it's just a place to put tools for a specific environment; i.e. PDO plus **PostgreSQL**. Finally we have to facade. The facade is just a collection of static methods that work right out of the box. All these functions use the toolbox and OODB to perform tasks without you having to know about the details. For more info Visit API pages.

Writing a Query Writer

A database driver for RedBeanPHP is called a Query Writer. The purpose of a query writer is to write and execute database specific SQL. The actual execution of a query is done by the database adapter. This adapter is passed to the writer so it can execute SQL statements directly; this means no additional wiring is required.

Interface

All Query Writers are expected to implement the RedBean_QueryWriter interface provided by RedBeanPHP. This interface describes the methods a Query Writer needs to implement. Because there is a decent amount of overlap between the writers it's probably a good idea to subclass the RedBean_QueryWriter_AQueryWriter abstract class. This class contains a lot of implementations for methods that are likely to work out of the box.

Constructor

When a Query Writer is created, a Database Adapter will be passed to its constructor. The constructor should therefore be able to receive a RedBean_Adapter instance. This adapter can then be stored in a property inside the Query Writer. The adapter can be used by the Query Writer to execute database specific SQL. This means that queries generated by the writer don't have to be returned to some object and no additional wiring is necessary. You can just pass the SQL to the exec-function of the adapter.

Tasks of a query writer

A query writer has to implement the RedBean_QueryWriter interface. More information on the Query Writer API can be found in the API documentation. The following methods need to be implemented:

addColumn
addConstraint
addFK
addIndex
addUniqueIndex
code
count
createTable
getTables
getValue
safeColumn
safeTable
scanType
selectRecord
sqlStateIn
updateRecord
widenColumn
wipe

Value types

To allow dynamic resize of database columns you need two methods: scanType() and code() as well as some class constants identifying different kinds of values. The scanType() method should be able to, given a certain value that needs to be stored in the database return the class constant that identifies the type of column to be used. Note that data types need to be ordinal; a higher type should not discard any information contained by a lower type. For instance it's fine to have types like: boolean,number,string but not boolean,date,number,string. This is because when converting date to number the field will lose information. The code() method is expected to work the other way around; given a database description of a column type it should return the class constant that maps. To prevent RedBeanPHP from 'shrinking' columns previously specified by a database engineer; we also implement a special constant named 'C_DATA_SPECIFIED'. This constant should have the highest ordinal numeric value (for instance 99). This causes manually changed columns to be regarded as the 'highest' type so they wont be altered back. After calling scanType, the OODB object will call getValue, this method has to return the value that has been scanned. In case of a special type you can return a modified value here. For instance if you scanned POINT(1 2) as being a spatial type, you have to return the binary insert value in getValue(). If you don't work with special types you can just return whatever has been passed to scanType. This default behaviour has already been implemented in AQueryWriter so if you extend that class, there is no need to write anything in this method except that in scanType you should copy the $value argument to $this->svalue;.

Basic CRUD methods

The job of basic CRUD methods is to write queries to INSERT, UPDATE, SELECT and DELETE rows from a table. There are two methods required to make this work: selectRecord() and updateRecord(). The selectRecord() method either selects or deletes a row. The updateRecord() method either inserts or updates a row. The selectRecord method accepts five parameters: $type, $conditions, $addSql=null, $delete=null, $inverse=false. The $type argument specified the type of bean the user is looking for. The $conditions argument is an array with conditions. Condition arrays are structured to be used in IN()-constructions. So, every entry in the array consists of a key and a subarray; here the key specifies the column that needs to be checked and the array of values represents the contents of the IN(...) in the SQL query. All entries are combined using the AND operator. $addSQL is used to specify additional SQL, your implementation should be able to integrate this SQL. If $delete is TRUE the user wants not just to select the row but delete it from the table. If $inverse is TRUE all non-matching rows should be selected/deleted. The updateRecord() method accepts 3 parameters: $type, $updateValues and $id. If $id equals 0, an INSERT should be performed. The array of update values should be structured like this: array($field=>$value, ...). Another CRUD method is wipe($type) which is used to quickly delete all rows in a table (or truncate).

SQL States

In RedBeanPHP not all SQL errors are bad. For instance querying a non-existant

column in fluid mode is common. However RedBeanPHP needs to know from your writer what a certain SQL state means, that is why your Query Writer must implement an sqlStateIn($state, $list) method. The $state argument contains the state provided by the database system while the list is a list of generic statements and RedBeanPHP wants to know whether the current state is somewhere in that list. The states RedBeanPHP cares about have been defined in the RedBean_QueryWriter interface: C_SQLSTATE_NO_SUCH_TABLE, C_SQLSTATE_NO_SUCH_COLUMN, C_SQLSTATE_INTEGRITY_CONSTRAINT_VIOLATION. RedBeanPHP will pass a database generated string of a certain SQL state along with one or more of these states and will ask your driver if it is one of these.

Scalability

Scalability is the ability of a system to handle growth. An application is scalable if it can be extended easily and support increasing numbers of users. There is no such thing as out-of-the-box scalability although many projects seem to claim this. Give me the most 'scalable' library in the world and I will make it crawl, simply by writing poor code. On the other hand, give me some clunky framework that does not scale at all and I can make it fly. A library or framework is only as scalable as the code that uses it allows it to be. Yes, it is as simple as that.

Does RB Scale?

There is no such thing as 'free scalability', you have to work for it. You have to adapt your application carefully to changing demands. So unlike other projects, I will give you an honest answer; RedBeanPHP does not offer 'free scalability' because it simply cannot predict the roadmap of your application. However, because RedBeanPHP is easy to use and easy to maintain, it will save you time and energy. This time and energy can then be used to focus on scalability and design proper solutions to allow your application to grow.

FUSE

There is one feature of RedBeanPHP that directly improves scalability though: FUSE. Fuse allows you build a prototype application with beans and controllers and add the models later on without having to refactor the old code. This works because models are connected to beans without having to change beans into models. Because of this there is a toll-free upgrade from prototype to full fledged application.

Arcane frameworks

Scalability is often used as an excuse by very arcane architectures like EJB. I therefore even consider it bad practice to take scalability into account at the start of a project, at least for most projects (talking about the typical web application here). I consider maintainability to be much more important.

Who uses RedBeanPHP

RedBeanPHP is used for all types of applications and tools. It has been integrated in various frameworks, it has been used for various projects ranging from financial business applications to embedded systems to measure changes in temperature.

RedBeanPHP is the ORM of choice of various cool frameworks.

Other

About

The mission of RedBeanPHP is to do ORM differently.

Easy, Agile, Fun

Data modelling should be easy, agile and fun. It should be easy, because code that no one can understand is barely maintainable. It should be agile because people tend to change their minds about almost everything, everyday. It should be fun, because what's life without fun? Why should you use industry standard tools instead of the tools that boost your productivity? Dare to choose for a different solution!

No more endless configurations

Instead of endless lines of configuration, RedBeanPHP requires no XML, YAML or INI files at all. Instead of switching back-and-forth between phpMyAdmin and your code, RedBeanPHP creates columns, tables and indexes on the fly. Instead of defining every detail about relations, RedBeanPHP lets you simply connect every object to every other object. Instead of writing queries using cumbersome query builders or strange SQL dialects, RedBeanPHP allows you to write your queries in plain old SQL. Instead of making your life miserable, RedBeanPHP seeks to inspire you to build great applications. Doing the dull work because you are not a machine...

RedBeanPHP makes your life easier by doing the dull work for you; creating databases (**SQLite**), creating tables, creating columns, resizing and adjusting columns, adding unique indexes to link tables, creating views, deploying databases and so on. RedBeanPHP supports every major open source database system out there; **MySQL**, **SQLite** and **Postgres**.

Allows your app to grow

RedBeanPHP is the only ORM layer that allows you to define models whenever you want (not upfront). All other ORM tools force you to define models before you even get a 'feel' of how the application works. With RedBeanPHP you have a chance to prototype your application first and then define the final models. While other ORMs

force you to refactor, RedBeanPHP allows your application to evolve in a natural way.

Open for the world

RedBeanPHP is open source, licensed both GPL and New BSD. RedBeanPHP is open, open for hobbyists, open for professionals, open for business, open for everyone. For you

You should not work for an ORM, an ORM should work for you. We try to deliver the best ORM ever made, because we believe we can and because we believe we have to and... because we believe you deserve it.

Extra

Visit RedBeanPHP Extra.

PDF Edition

Besides the online version of the manual we offer a PDF version of the manual for printing. download PDF version.