Connecting Beans

A Guide to Implementing Basic Data Relationships in RedBeanPHP

Table of Contents

About This Guide

This Is a Quick Reference Only

RedbeanPHP is one of the most innovative ORM libraries for PHP to appear in quite some time. It is also extremely easy to use, but its different approach to data mapping can be puzzling or confusing for the newcomer, who might be used to modeling data relationships according to more ‘traditional’ patterns.

In particular, the philosophy of RedbeanPHP is that you should not have to change your object model in order to fit the requirements of data storage. You should code you models as your object model requires, and the database tables should be automatically created to store your object data, and their relationships to each other, as needed. This is a very pragmatic approach, capable of drastically reducing data design and programming time. It can, however, have a couple of drawbacks.

If the programmer is porting an existing codebase to RedbeanPHP, then often they need to use existing data from an already setup database. Or perhaps the programmer is an expert on SQL, and is used to modelling their data first in their development process. RedbeanPHP is certainly suitable for these kinds of projects and development styles, but in these instances you need to understand how to use the different RedbeanPHP objects and methods to map the traditional data relationships in your own system.

This guide intends to show **some** of the different strategies available in RedbeanPHP to model all basic data relationships between database tables. So when you need figure out how to setup a ‘many-to-many’ or ‘self-referential one-to-many’ relationship between your objects in RedbeanPHP that access your database tables in a certain way, you can use this guide as a starting point, and save yourself some thinking time!

Read the RedbeanPHP Manual

RedbeanPHP already has an excellent - and concise - online manual, as well as an extensively documented API. This guide will only cover a very small part of the material presented in the manual, and only as it is necessary to explain the code presented here.

This is not intended as a guide to using RedbeanPHP. In order to use RedbeanPHP effectively, and make use of all its wonderful methods and shortcuts, you must read the official manual.

The Code In This Guide Is Bad

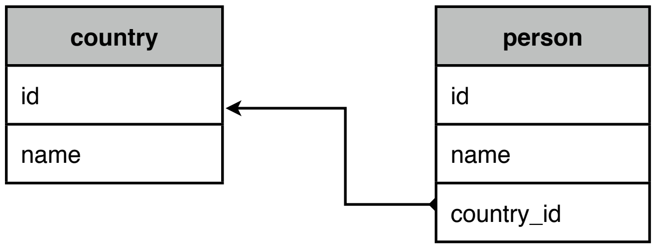
The code in this guide is usable, but it is inefficient, and does not follow all recommended practices of good object-oriented design, or of efficient data-modelling. Specifically, it does not show you how to setup model objects in RedbeanPHP, or how to write good, concise and customised accessor methods - all of which could make working with complex relationships a lot easier. The code here is intended merely to illustrate how you can use RedbeanPHP to setup and access data in the database so your tables and relationships end up conforming to certain data relationships.

Simple Relationships

Simple One-to-Many

Definition

A *simple one-to-many* relationship is a relationship between 2 different types of objects. Each 'one' object may be related to several 'many' objects, but each 'many' object may only be related to a single 'one' object. The database tables of a one-to-many relationship often look like this:



In this system, a person must indicate in which country they currently live. Each *country* will have many *persons* who live there, but each *person* will only ever be living in one *country*.

It is, therefore, a one-to-many relationship: *country* is the 'one', and *person* is the 'many'.

A *country_id* field is created in the 'person' table to track in which country the person is living. This field establishes the relationship between the tables in the database.

Redbean Example

We are building a system to manage a lending library for our local high school. The library will need to keep track of which books are being borrowed by which students. Each book may only be borrowed by one student at a time. Each student, however, will be allowed to borrow several books at once, if they wish. This is a simple one-to-many relatioship.

Creating the Objects

Using *R::dispense()*, create the student and book objects, populating them with properties as required. Use *R::store()* to save the objects to the database. Redbean will create the 'student' and 'book' tables automatically for you, if they don't already exist.

```
<?php
        // creating a ‘student’ object to be stored in a table named ‘student’:
        $student = R::dispense(‘student’);

        // adding properties to ‘student’, as needed:
        $student->name = “John Smith”;
        $student->email = “j.smith@email.com”;
        //... and so on

        // creating 3 ‘book’ objects, to be stored in a table named ‘book’:
        list($book1, $book2, $book3) = R::dispense(‘book’, 3);

        //  adding properties as needed:
        $book1->title = “Foundation Maths”;
        $book1->author = “A. Blah”;
        //... and so on, for each book

        // store ‘student’ and ‘book’ objects to save them in the database:
        R::store($student); R::store($book1); R::store($book2); R::store($book3);
?>
```

Establishing the Relationships
Once the objects are created, we can establish the relationship between them. In this example, we want to establish that a student is borrowing 3 books from the library. There are 2 basic ways to approach this.

• *Option 1)* Link The Student to Each Borrowed Book
We can create a 'student' property in each book, and this property will point to the borrowing student. In Redbean this is called *nesting* the student in the book:

```
<?php
      // nesting $student in $book1, $book2, and $book3:
      $book1->student = $student;
      $book2->student = $student;
      $book3->student = $student;

      // store 'book' objects to save the relationships in the database:
      R::store($book1); R::store($book2); R::store($book3);
?>
```

• *Option 2)* Link Books to Borrowing Student
Alternatively, we can link an entire array of books to a student at once, using a magic *ownType* array property - it links an array of that *type* of object automatically for you:

```
<?php
      // once $student and $book1, $book2, $book3 have been created:
      $student->ownBook = array($book1, $book2, $book3);

      // store 'student' object to save the relationships in the database:
      R::store($student);
?>
```

Whichever method you choose to use, the *R::store()* method will automatically add a 'student_id' field to the book table, and store the one-to-many relationship between objects, as expected.

Accessing the Related Data

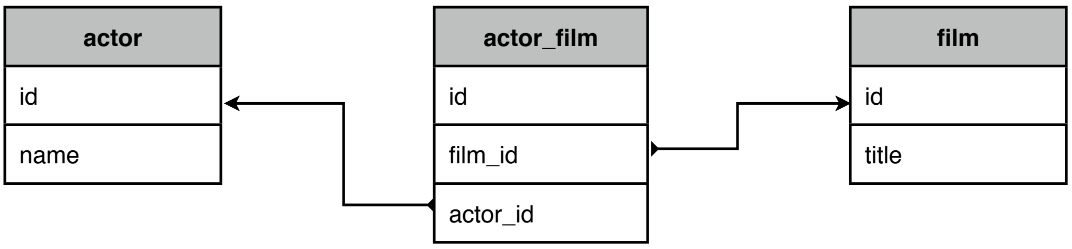
```
<?php
      // accessing the student data from a book object:
      echo "Book1 was borrowed by {$book1->student->name}.";

      // accessing the complete array of linked books from a student object:
      $books = $student->ownBook;

      // accessing individual properties in linked books from a student object:
      $title = $student->ownBook[1]->title; // array key = table's primary key!
?>
```

Important: The key for each array element in the *ownType* array is **not** a numerical index: it is the primary key for that record as used by its table in the database.

Simple Many-to-Many

Definition

A *simple many-to-many* relationship is a relationship between 2 different types of objects, where an object on either side of the relationship may be related to many objects on the opposite side. The usual way to store this type of relationship in database tables is:



In this system, we must store which actors have worked in which films. An *actor* may have worked in several *films*, and each film will have several actors. It is, therefore, a many-to-many relationship.

In most database systems, an intermediate table will be created, which will store the links between actors and films. Each record in this in-between table will have an *actor_id* field, and a *film_id* field to establish the individual link between these records.

Redbean Example

We need to setup a system to help our local high school keep track of which students attend which courses. This is a simple many-to-many relationship: each course can have several students, and each student may be attending several courses.

Creating the Objects
Using *R::dispense()*, create the student and course objects, populating them with properties as required. Use *R::store()* to save the objects to the database. Redbean will create the 'student' and 'course' tables automatically for you, if they don't already exist.

```
<?php
      // creating 3 ‘student’ objects, to be stored in a table named ‘student’:
      list($student1, $student2, $student3) = R::dispense(‘student’, 3);

      // adding properties to each ‘student’, as needed:
      $student1->name = “John Smith”;
      $student1->email = “j.smith@email.com”;
      //... and so on

      // store objects to save in the database:
      R::store($student1); R::store($student2); R::store($student3);

      // creating 2 ‘course’ objects, to be stored in a table named ‘course’:
      list($course1, $course2) = R::dispense(‘course’, 2);

      //  adding properties as needed:
      $course1->title = “Foundation Maths”;
      $course1->start_date = “2012-03-19”;
      //... and so on, for each course

      // store objects to save in the database:
      R::store($course1); R::store($course2);
?>
```

Establishing the Relationships
Once course and student objects have been created, we can establish connections between them. In this example we have 2 courses, and 3 students. We want to establish relationships that will show that students 1 and 2 attend course A, and students 2 and 3 attend course B.

• *Option 1)* Using the Magic *sharedType* Property

```
<?php
      // linking students to courses - adding the entire array at once:
      $course1->sharedStudent = array($student1, $student2);
      $course2->sharedStudent = array($student2, $student3);

      // store ‘course’ objects to save the relationship in the database:
      R::store($course1); R::store($course2);
?>
```

We can also establish the connections in reverse:

```
<?php
      // linking courses to students - adding to the array one by one:
      $student1->sharedCourse[] = $course1;
      $student2->sharedCourse[] = $course1;
      $student2->sharedCourse[] = $course2;
      $student3->sharedCourse[] = $course2;

      // store ‘student’ objects to save the relationship in the database:
      R::store($student1); R::store($student2); R::store($student3);
?>
```

The *sharedType* magic property establishes relationships between objects, and this relationship is stored in a separate table in the database. The *R::store()* method, when called with an object that has a *sharedType* property, will create the appropriate in-between table, if it doesn't already exist - in this case a table named 'course_student' - and will store the many-to-many relationships between objects in that table, as expected.

• *Option 2)* Using the *R:associate()* Method

```
<?php
      // once students and courses have been created:
      R::associate($course1, $student1); // or R::associate($student1, $course1)
      R::associate($course1, $student2);

      R::associate($course2, $student2);
      R::associate($course2, $student3);
?>
```

If you use the *R::associate()* method, there is no need to call *R::store()*. It will automatically create the appropriate in-between table in the database, if it doesn't already exist, and will store the many-to-many relationships between objects in that table, as expected.

Accessing the Related Data

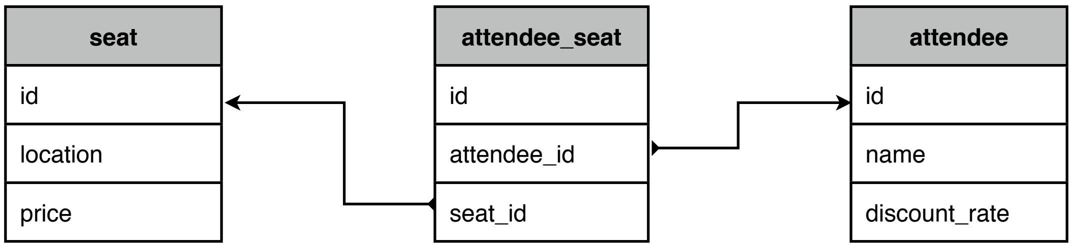
```
<?php
      // accessing the array of students related to Course 1:
      $students = R::related($course1, 'student'); // using R::related()

      // accessing the array of students related to Course 2:
      $students = $course2->sharedStudent[]; // using magic sharedType array

      // accessing the array of courses related to Student 1:
      $courses = R::related($student1, 'course'); // using R::related()

      // accessing the array of courses related to Student 2:
      $courses = $student2->sharedCourse[]; // using magic sharedType array
?>
```

Important: The key for each array element in the *sharedType* array is **not** a numerical index: it is the primary key for that record as used by its table in the database.

Simple One-to-One

Definition

A *simple one-to-one* relationship is a relationship between 2 different types of objects, where an object on either side of the relationship may be related to only one single object on the opposite side. In most systems these relationships are rare, and needed only in very specific situations - ie., for integration of data from external systems, for modularity and reuse of certain code and objects, or for security reasons. A typical way to store this type of relationship in database tables is to treat it as a simple many-to-many:



In this system, we will track seating arrangements for attendees at a certain event. During this event, each attendee will be given a seat. Each seat, therefore, will be occupied by a single attendee. We could keep all the information about seats inside the *attendee* object itself. We could alternatively keep all the attendee information inside the seat object. But we will need to use *attendee* and *seat* objects in other projects. So we decide to have separate objects for *attendees* and their *seats*, and establish a simple one-to-one relationship between them.

By creating an in-between table between seats and attendees we help keep the encapsulation of *seat* and *attendee* objects - there wil be no 'seat' property in attendees, and no 'attendee' property in seats. These objects can then be included in other projects with no dependencies.

Redbean Example

We have a system where we are required to keep some private information about our users, and we must keep that information securely. The information, in this case, will be credit card details, which will be kept as encrypted hashes in our database. In our system a user is only allowed to keep details of a single credit card. We could simply add a 'credit_card' property to our 'user' object, and store that information in a 'user' database table, along with all other information for the user. This solution is not very modular, and arguably less secure than keeping separate objects and tables for *users* and *credit_cards*.

By using separate user and credit_card objects, we can make both objects more reusable in other projects - ie., the credit_card object could be connected to 'company' objects in a different project, and to 'bank_account' objects in a financial system. It also means that we can keep all security code in the 'credit_card' class, where it belongs, and not in the 'user'.

Important: For almost all applications, storing credit card information is a **very bad idea**, and poses a high security risk to both your users and you. The example is used here merely to illustrate that this type of relationship may be necessary sometimes due to security issues - ie., we may not wish to keep a user's credit card information together with the user's record, for security.

Creating the Objects
Using *R::dispense()*, create the *card* and *user* objects, populating them with properties as required. Use *R::store()* to save the objects to the database. Redbean will create the ‘card’ and ‘user’ tables automatically for you, if they don’t already exist.

```
<?php
	// creating a ‘card’ object, to be stored in a table named ‘card’:
	$card = R::dispense(‘card’);

	// adding properties to card, as needed:
	$card->type = “Visa”;
	$card->expiry = “2014-11”;
	//...and so on

	// creating a ‘user’ object, to be stored in a table named ‘user’:
	$user = R::dispense(‘user’);

	// adding properties to user, as needed:
	$user->name = “John Smith”;
	$user->email = “j.smith@email.com”;
	//... and so on

	// store objects to save in the database:
	R::store($card); R::store($user);
?>
```

Establishing the Relationships
Once ‘card’ and ‘user’ objects have been created, we can easily establish the relationship between them, using *R::associate()*:

```
<?php
	// once user and card have been created:
	R::associate($user, $card);
?>
```

Using the *R::associate()* method, there is no need to call *R::store()*. It will automatically create the appropriate in-between table in the database, if it doesn’t already exist, and will store the many-to-many relationships between objects in that table, as expected.

Accessing the Related Data
To access the related object, we can use the *R::relatedOne()* method. This method returns a single object associated through the in-between table:

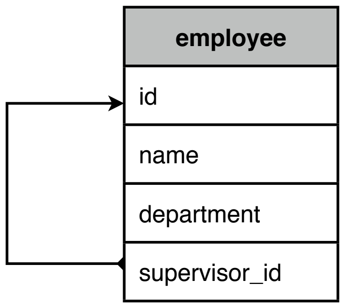
```
<?php
	// accessing card related to a given user:
	$card = R::relatedOne($user, ‘card’);

	// accessing user related to a given card:
	$user = R::relatedOne($card, ‘user);
?>
```

Self-Referential Relationships

Self-Referential One-to-Many

Definition

A *self-referential one-to-many* relationship is a relationship between 2 objects that are of the same type, where each 'one' object may be related to several 'many' objects, but each 'many' object may only be related to a single 'one' object. A database table containing a self-referential one-to-many relationship may look like this:



In this system, we are tracking employees, and the supervisor to whom they must report. An employee always reports to only one supervisor. Each supervisor, however, will get reports from several employees. This would be a simple one-to-many relationship, but 'supervisors' and 'employees' are both just *employees* - they are the same type of object. This makes this a self-referential relationship.

A *supervisor_id* field is created in the 'employee'. This field establishes the relationship between an 'employee' and the 'supervisor', who is also an employee in the same table.

Redbean Example

We have are building a new website where content is organised in a hierarchy of pages: a page can have many children pages, and each child page can have many pages of its own. Each page, of course, can only ever have one 'parent' page. This is a one-to-many relationship, and as both sides of the relationship are the same object type (pages), then this is a self-referential relationship.

Creating the Objects
Using *R::dispense()*, create *page* objects, populating them with properties as required. Use *R::store()* to save the objects to the database. Redbean will create the 'page' table automatically for you, if it doesn't already exist.

```
<?php
	// creating 3 ‘page’ objects, to be stored in a table named ‘page’:
	list($page1, $page2, $page3) = R::dispense(‘page’, 3);

	// adding properties to pages, as needed:
	$page1->title = “About Us”;
	$page1->description = “company and contact information”;
	//... and so on, for all ‘pages’

	// store objects to save in the database:
	R::store($page1); R::store($page2); R::store($page3);
?>
```

Establishing the Relationship
Once pages have been created, we can establish the ‘parent’ relationship between them. In our example, let’s say we have a page1, who is the parent of 2 other pages: page2 and page3.

To establish this relationship, we can simply create a ‘parent’ property for each page, and point this property to the parent page. In Redbean this is called *nesting* one page into another:

```
<?php
      // nesting page1 into pages 2 and 3:
      $page2->parent = $page1;
      $page3->parent = $page1;

      // store objects to save in the database:
      R::store($page2); R::store($page3);
?>
```

The *R::store()* method will automatically add a ‘parent_id’ field to the ‘page’ table, if it doesn’t already exist, and store the primary key of the related page there, as expected.

Accessing the Related Data
If we try to retrieve the ‘parent’ of a page using *$page->parent*, we will get an error. This is because Redbean will assume this to be a simple one-to-many relationship, and it will try to look for a table named ‘parent’, which does not exist. We must tell Redbean that although we have called this a ‘parent’, it really should fetch the record from the table ‘page’ itself. We do this using the *fetchAs()* method:

```
<?php
      // accessing the parent page, given a child $page2:
      $parent = $page2->fetchAs(‘page’)->parent;
?>
```

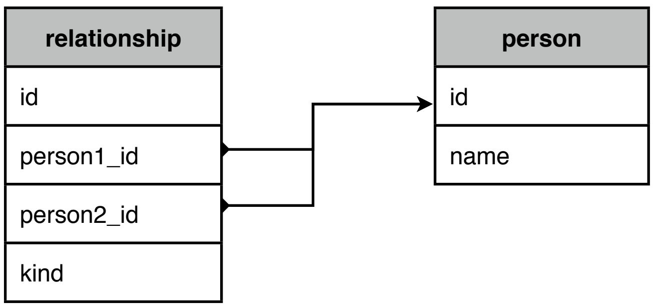
To retrieve an array of all children pages for a certain parent, we must use the *R::find()* method, to perform a quick query in the table:

```
<?php
      // retrieving an array of all children pages, given parent $page1:
      $children = R::find(‘page’, ‘ parent_id = ?’, array($page1->getID()));
?>
```

Self-Referential Many-to-Many

Definition

A *self-referential many-to-many* relationship is a relationship between 2 objects of the same type, where an object on either side of the relationship may be related to many objects on the opposite side. The usual way to store this type of relationship in database tables is:



In this system, we keep track of family relationships between people. Each person may be related to any number of person in the database. This is, therefore, a many-to-many relationship between persons. As *person* objects are used on both sides of the 'many' relationship, it is self-referential.

A separate table is created to keep track of each familial relationship between 2 people. Note that the ids of both people in the relationship will point to (different) records in the same 'person' table.

Redbean Example

We have an online club where members can become 'friends' with other members. A member can have any number of friends, and can become a friend to any number of members. This is a many-to-many relationship between friends, who are really *members*. Both sides of the relationship are comprised of the same type of object (members), so this makes it a self-referential relationship.

Creating the Objects
Using *R::dispense()*, create *member* objects, populating them with properties as required. Use *R::store()* to save the objects to the database. Redbean will create the 'member' table automatically for you, if it doesn't already exist.

```
<?php
	// creating 3 'member' objects, to be stored in a table named 'member':
	list($member1, $member2, $member3) = R::dispense('member', 3);

	// adding properties to members, as needed:
	$member1->name = "J. Smith";
	$member1->email = "j.smith@email.com";
	//...and so on, for all members, as required

	// store objects to save in the database:
	R::store($member1); R::store($member2); R::store($member3);
?>
```

Establishing the Relationship
Once members have been created, we can establish the 'friendship' relationship between them. In our example, let's say we have 3 members who are reciprocal friends - each one is a 'friend' of the other 2.

In order to establish these friendships, we will use *R::dispense()* to create separate ‘friendship’ objects (and a separate ‘friendship’ database table), which will link 2 members as friends, using *nesting*:

```
<?php
      // once members have been created,
      // create a ‘friendship’ between 2 members & store in table ‘friendship’:
      $friendship1 = R::dispense(‘friendship’);
      $friendship1->friendA = $member1;
      $friendship1->friendB = $member2;

      // create more ‘friendships’, as needed:
      $list($friendship2, $friendship3) = R::dispense(‘friendship’, 2);
      $friendship2->friendA = $member1;
      $friendship2->friendB = $member3;
      $friendship3->friendA = $member2;
      $friendship3->friendB = $member3;

      // store ‘friendship’ objects to save the relationships in the database:
      R::store($friendship1); R::store($friendship2); R::store($friendship3);
?>
```

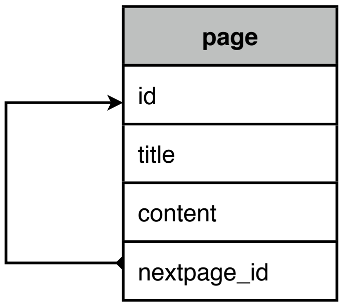
Accessing the Related Data
In order to get an array containing all of a member’s friends, we will perform 2 queries using *R::find()*. In the first query we will get all ‘friendships’ in which the member is a part, either as ‘friendA’ or ‘friendB’. Then, we query back the ‘member’ table in order to get a related member for each friendship record found:

```
<?php
      // accessing the array of friends for a given member,
      // in this case, $member1:

      // first, get all ‘friendships’ connected to this member:
      $id = $member1->getID();
      $query = “ friendA_id = $id OR friendB_id = $id”;
      $friendships = R::find(‘friendship’, $query);

      // then, get all paired ‘members’ connected to those friendships:
      foreach($friendships as $friendlink) {
            if ($id == $friendlink->friendA_id) {
                  // if $member1 is friendA, then we get friendB:
                  $query = ‘ id = ‘ . $friendlink->friendB_id; }
            else {
                  // if $member1 is friendB, then we get friendA:
                  $query = ‘ id = ‘ . $friendlink->friendA_id; }

            // finally, the related friend is added to a ‘friends’ array:
            $friends[] = R::findOne(‘member’, $query);
      }
?>
```

Self-Referential One-to-One

Definition

A *self-referential one-to-one* relationship is a relationship between 2 objects of the same type, where an object on either side of the relationship may be related to only one single object on the opposite side. This type of relationship usually implies some form of 'ordered chain' between the objects: object a somehow follows object b, which follows c, and so on. A typical way to store this type of relationship in a database table is:



This system contains several pages which are linked sequentially, indicating the reading order. A page can only be followed by one other page, and it can only follow another page. This would be a self-referential one-to-one relationship.

A *nextpage_id* field has been created in the 'page' table. This field establishes the relationship between a 'previous' and a 'next' page in the same table.

Redbean Example

We are setting up an online system for our local high school, where there is a 'buddy' program to help integrate new students. Each new student arriving at the school is given a 'buddy': an existing student who will provide support and assistance during the first few weeks, and help the new student settle into the school.

The school policy is that each new student is allocated one single 'buddy', and a student cannot be a buddy to more than a single new student at any given time. This describes a one-to-one relationship between new students and buddies. As both sides of the relationship are the same type of object (students), then it is a self-referential relationship.

Creating the Objects

Using *R::dispense()*, create *student* objects, populating them with properties as required. Use *R::store()* to save the objects to the database. Redbean will create the 'student' table automatically for you, if it doesn't already exist.

```
<?php
	// creating 2 ‘student’ objects, to be stored in a table named ‘student’:
	list($oldstudent, $newstudent) = R::dispense(‘student’, 2);

	// adding properties to students, as needed:
	$oldstudent->name = “John Smith”;
	$oldstudent->email = “j.smith@email.com”;
	$newstudent->name = “Bob Blah”;
	$newstudent->email = “bob@someplace.com”;

	// store objects to save in the database:
	R::store($oldstudent); R::store($newstudent);
?>
```

Establishing the Relationship
Once new and existing students have been created, we can establish the ‘buddy’ relationship between them. In our example, OldStudent is going to be a buddy to NewStudent.

To establish this relationship, we can simply create a ‘buddy’ property for each student, and point this property to the buddy. In Redbean this is called *nesting* one student into another:

```
<?php
      // nesting one student into another, as a ‘buddy’:
      $newstudent->buddy = $oldstudent;

      // store object to save the relationship in the database:
      R::store($newstudent);
?>
```

The *R::store()* method will automatically add a ‘buddy_id’ field to the ‘student’ table, if it doesn’t already exist, and store the primary key of the related student there, as expected.

Accessing the Related Data
If we try to retrieve the ‘buddy’ of a student using *$student->buddy*, we will get an error. This is because Redbean will assume this to be a simple one-to-many relationship, and it will try to look for a table named ‘buddy’, which does not exist. We must tell Redbean that although we have called the object we are trying to fetch a ‘buddy’, it really should fetch it from the table ‘student’ itself. We do this using the *fetchAs()* method:

```
<?php
      // accessing the buddy, given the student:
      $buddy = $newstudent->fetchAs(‘student’)->buddy;
?>
```

To retrieve a new student that an old student is being a buddy to, we must use the *R::find()* method, to perform a quick query in the ‘student’ table:

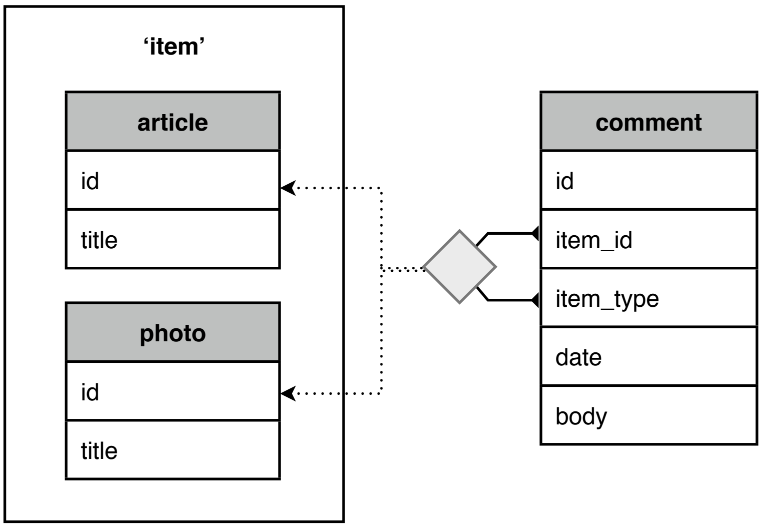
```
<?php
      // accessing the newstudent that the oldstudent is being a buddy to:
      $student = R::findOne(‘student’, ‘ id = ?’, array($oldstudent->getID()));
?>
```

Polymorphic Relationships

PolyOne-to-Many

Definition

A poly*one-to-many* relationship is similar to a simple one-to-many relationship, but here the type of the 'one' object can vary, while the 'many' object is of a fixed type. The 'one' object is therefore said to be 'polymorphic'. The database tables of a polyone-to-many relationship may look like this:



This system allows users to attach comments to different 'items'. An item may be an article, or a photo. Each item (article or photo) may have several comments connected to it, but each comment is attached to only a single article or photo. We have a one-to-many relationship: one item for many comments. The items, however, can be different types of objects (articles or photos). This makes it a polymorphic relationship on the 'one' side: a polyone-to-many.

As can be seen, this relationship is similar to a simple one-to-many, but an extra 'item_type' field has to be used to help select the table to which the item belongs - ie. the type of the item..

Redbean Example

We are setting up a management system for the lending library of our local high school. Both teachers and students are allowed to borrow books from the library. Teachers and students are, therefore, classified as 'borrowers'. Each borrower can borrow many books, if they so wish. Each book, however, can only be borrowed by one borrower at a time. This establishes a one-to-many relationship between borrowers (on the 'one' side) and books (on the 'many' side).

As a borrower can be more than one type of object - in our case, a teacher or a student - this is a polymorphic relationship, on the 'one' side.

Creating the Objects
Using *R::dispense()*, create *book, teacher* and *student* objects, populating them with properties as required. Use *R::store()* to save the objects to the database. Redbean will create the 'book', 'teacher' and 'student' tables automatically for you, if needed.

```
<?php
      // creating 3 ‘book’ objects, to be stored in a table named ‘book’:
      list($book1, $book2, $book3) = R::dispense(‘book’, 3);

      // adding properties to books, as needed:
      $book1->title = “Foundation Maths”;
      $book2->title = “World Atlas”;
      $book3->title = “English Grammar”;

      // store objects to save them in the database:
      R::store($book1); R::store($book2); R::store($book3);

      // creating a ‘teacher’ and a ‘student’ object - our ‘borrowers’:
      $teacher = R::dispense(‘teacher’);
      $teacher->name = “John Smith”;

      $student = R::dispense(‘student’);
      $student->name = “Bob Blah”;

      // store objects to save them in the database:
      R::store($teacher); R::store($student);
?>
```

Establishing the Relationships
Once teachers, students and books have been created, we can establish connections between them. In this example, we have 1 teacher, 1 student and 3 books. The teacher borrowed 1 book, and the student borrowed the remaining 2.

Unfortunately, we cannot simply *nest* the borrower in the book, as we would be able to do for a simple one-to-many relationship. Because our borrower is a polymorphic object, we must keep track not just of its id, but also of its type. We can use the *getID()* method to find out the borrower’s id, and the *getMeta(‘type’)* method to find out its type, and then explicitly store this information in the database:

```
<?php
      // assign ‘borrower’ id+type information to link it to a book:
      $book1->borrower_id = $teacher->getID();
      $book1->borrower_type = $teacher->getMeta(‘type’);
      $book2->borrower_id = $student->getID();
      $book2->borrower_type = $student->getMeta(‘type’);

      // note that we don’t have to know the type of the borrower:
      $borrower = $student; // could just as well be $teacher
      $book3->borrower_id = $borrower->getID();
      $book3->borrower_type = $borrower->getMeta(‘type’);

      // store ‘book’ objects to save the relationships in the database:
      R::store($book1); R::store($book2); R::store($book3);
?>
```

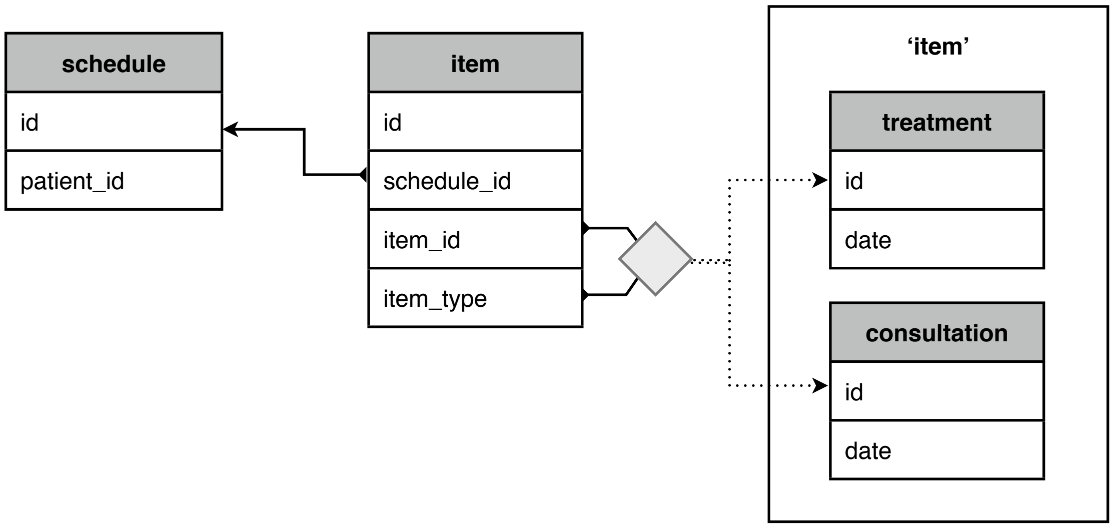
Accessing the Related Data
We need to perform a very simple query in order to obtain a book’s borrower, or to get an array of books given any borrower:

```
<?php
      // accessing related borrower from book:
      $type = $book1->borrower_type;
      $id = $book1->borrower_id;
      $borrower = R::findOne($type, ‘ id = ?’, array($id));

      // accessing all books related to a borrower:
      $values[‘id’] = $borrower->getID(); // could be ‘student’ or ‘teacher’
      $values[‘type’] = $borrower->getMeta(‘type’);
      $books = R::find    ( ‘book’,
                          ‘ borrower_id = :id AND borrower_type = :type’,
                          $values );
?>
```

One-to-PolyMany

Definition

A *one-to-polymany* relationship is similar to a simple one-to-many relationship, but here the 'one' object is of a fixed type, while the type of the 'many' object can vary. The 'many' object is therefore said to be 'polymorphic'. The database tables of a one-to-polymany relationship usually look like this:



This system keeps medical treatment schedules for patients. Each treatment schedule is made up of a series of treatments, followed by periodic consultations with the doctor. Each schedule, therefore, will have several items (treatments and consultations), but each item (treatment or consultation) can only be allocated to a single schedule. This characterises a one-to-many relationship: one schedules has many items. The items, however, can be different types of objects (treatments or consultations). This makes it a polymorphic relationship on the 'many' side: a one-to-polymany.

An in-between 'item' table is necessary to keep the relationship between schedules and their items. This table must keep not just the item id, but also the item type.

Redbean Example

We are setting up a management system for the lending library of our local high school. Students are allowed to borrow books and DVDs from the library. Each student can borrow several items (books and DVDs) at once, if they so wish. Each item, however, can only be lent to a single student. This establishes a one-to-many relationship between students on the 'one' side and items (books and DVDs) on the 'many' side.

As items can be more than one type of object - in this case, a book or a DVD - then this is a polymorphic relationship, on the 'many' side.

Creating the Objects
Using *R::dispense()*, create *book, dvd* and *student* objects, populating them with properties as required. Use *R::store()* to save the objects to the database. Redbean will create the 'book', 'dvd' and 'student' tables automatically for you, if they don't already exist.

```
<?php
      // creating 2 ‘student’ objects, to be stored in a table named ‘student’:
      list($student1, $student2) = R::dispense(‘student’, 2);

      //adding properties to students, as needed:
      $student1->name = “John Smith”;
      $student2->name = “Bob Blah”;

      // store objects to save in the database:
      R::store($student1); R::store($student2);

      //...repeat as above for ‘book’ and ‘dvd’ objects.
?>
```

Establishing the Relationships
In this example, let’s say we have 2 students borrowing 2 DVDs and 1 book. The first student borrowed 1 DVD, and the second borrowed a DVD and a book. Once students, books and DVDs have been created, we can connect them.

We do not, however, wanto to link our ‘student’ table directly to the book and dvd tables. Rather, we want to have a to generic ‘item’ table in between, and link our students to that - view diagram at the beginning of this chapter. Establishing the connections between students and books/dvds, therefore, will be a 2-stage process: first, we must create a generic ‘item’ object for each book or dvd to be borrowed, and then we link the student to the ‘items’ being borrowed using the magic *ownType* property - as we do in a simple one-to-many relationship:

```
<?php
      // once students, books and DVDs have been created,
      // create ‘item’ objects for all book and dvds:
      list($item1, $item2, $item3) = R::dispense(‘item’, 3);

      $item1->item_id = $dvd1->getID();
      $item1->item_type = $dvd1->getMeta(‘type’);

      $item2->item_id = $dvd2->getID();
      $item2->item_type = $dvd2->getMeta(‘type’);

      $item3->item_id = $book->getID();
      $item3->item_type = $book->getMeta(‘type’);

      // store item objects to save relationships in database:
      R::store($item1); R::store($item2); R::store($item3);

      // now, we can use the magic ownType property
      // to associate students with items being borrowed
      $student1->ownItem = array($item1); // could also use R::associate()
      $student2->ownItem = array($item2, $item3);

      // store student objects to save relationships in database:
      R::store($student1); R::store($student2);
?>
```

Accessing the Related Data

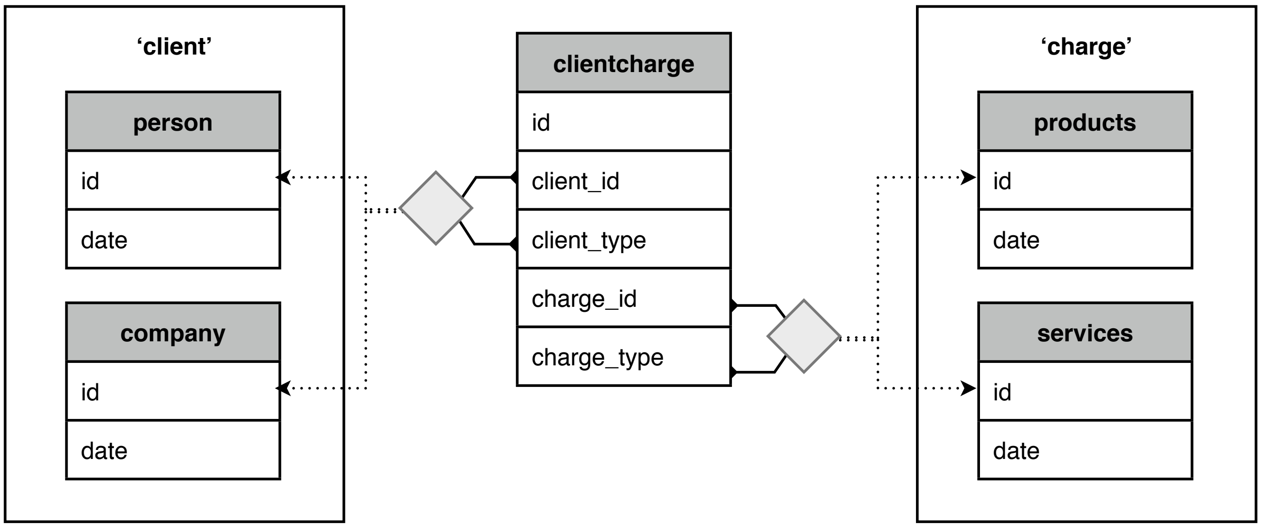
Because of the in-between ‘item’ table, we will need to perform a couple of queries in order to find related objects. The first query will be to find the related object in the ‘item’ table, and the second to find the objects related to the ‘item’ object in the related table:

```
<?php
	// accessing the borrowing student, given a book or dvd:
	// first, get the ‘item’ object for the book or dvd:
	$type = dvd1->getMeta(‘type’);
	$id = dvd1->getID();
	$item = R::findOne(‘item’,
	                    ‘ item_id = :id AND item_type = :type’
	                    , array($id, $type)));
	// then, get the ‘student’ related to that ‘item’:
	$student = R::relatedOne($item, ‘student’); // could also use R::findOne()

	// accessing an array of borrowed books and dvds for a given student:
	// to get an array of ‘items’ related to a ‘student’ we use ownItem.
	// we then use this array to fetch all related books and dvds:
	foreach($student->ownItem as $item) {
		$key = $item->getID();
		$items[$key] = R::find($item->item_type,
		                    ‘ id = ?’,
		                    array($item->item_id));
	}
	// $items will have all books and dvds borrowed by the related student
?>
```

PolyOne-to-PolyMany

Definition

A *polyone-to-polymany* relationship is similar to a simple one-to-many relationship, but here the types of the objects on both sides of the relationship is not fixed, and can vary. Both the 'one' and the 'many' objects are therefore said to be 'polymorphic'. The database tables of a polyone-to-polymany relationship can typically look like this:



This system keeps track of charges incurred by clients. We have 2 types of clients: *persons* and *companies*. We supply these clients with *products*, as well as with *services,* for which we charge. A client, therefore, can have several charges for products and services. Each charge, however, will be allocated to a single client, which will be a person or a company. This characterises a one-to-many relationship between clients and charges. Both clients and charges, however, can be different types of objects: clients can be *persons* or *companies*, and charges can be *products* or *services*. This makes it a polymorphic relationship on both the 'one' and the 'many' sides.

An in-between 'item' table is necessary to keep the relationship between client and charge, and it has to store the ids and types for both the client and the charge.

Redbean Example

We are setting up a management system for the lending library of our local high school. Teachers and students are allowed to borrow books and DVDs from the library. Each borrower (teacher or student) can borrow several items (books and DVDs) at once, if they so wish. Each item, however, can only be connected to a single borrower. This establishes a one-to-many relationship between borrowers (teachers and students) on the 'one' side and items (books and DVDs) on the 'many' side.

As both borrowers and items can be more than one type of object, then this is a polymorphic one-to-many relationship, on both sides.

Creating the Objects

Using *R::dispense()*, create *book, dvd, teacher* and *student* objects, populating them with properties as required. Use *R::store()* to save the objects to the database. Redbean will create the 'book', 'dvd', 'teacher' and 'student' tables automatically for you, if they don't already exist.

```
<?php
      // creating a ‘student’ objects, to be stored in a table named ‘student’:
      $student= R::dispense(‘student’);
      $student->name = “John Smith”; // add properties as needed

      // creating a ‘teacher’ objects, to be stored in a table named ‘teacher’:
      $teacher = R::dispense(‘teacher’);
      $teacher->name = “Bob Blah”; // add properties as needed

      // store objects to save in the database:
      R::store($student); R::store($teacher);

      //...repeat as above for ‘book’ and ‘dvd’ objects.
?>
```

Establishing the Relationships
Once teachers, students, books and DVDs have been created, we can establish connections between them. In this example, let’s say we have a teacher who is borrowing one DVD, and a student who is borrowing a DVD and a book.

Establishing the connections between students/teachers and books/dvds will be a 2-stage process: first, we must create a generic ‘item’ object representing each book or dvd to be borrowed, and then we link the student or teacher to the ‘items’ being borrowed:

```
<?php
      // once teacher, student, book and DVDs have been created,
      // create ‘item’ objects to represent books and dvds being borrowed:
      list($item1, $item2, $item3) = R::dispense(‘item’,3);

      // link item to a dvd or book by storing its id+type:
      $item1->item_id = $dvd1->getID();
      $item1->item_type = $dvd1->getMeta(‘type’);

      $item2->item_id = $dvd2->getID();
      $item2->item_type = $dvd2->getMeta(‘type’);

      $item3->item_id = $book->getID();
      $item3->item_type = $book->getMeta(‘type’);

      // now, link the borrowers to the items by storing their id+type:
      $item1->borrower_id = $teacher->getID();
      $item1->borrower_type = $teacher->getMeta(‘type’);

      $item2->borrower_id = $student->getID();
      $item2->borrower_type = $student->getMeta(‘type’);

      $item3->borrower_id = $student->getID();
      $item3->borrower_type = $student->getMeta(‘type’);

      //...repeat for item3

      // store ‘item’ objects to save relationships in database:
      R::store($item1); R::store($item2); R::store($item3);
?>
```

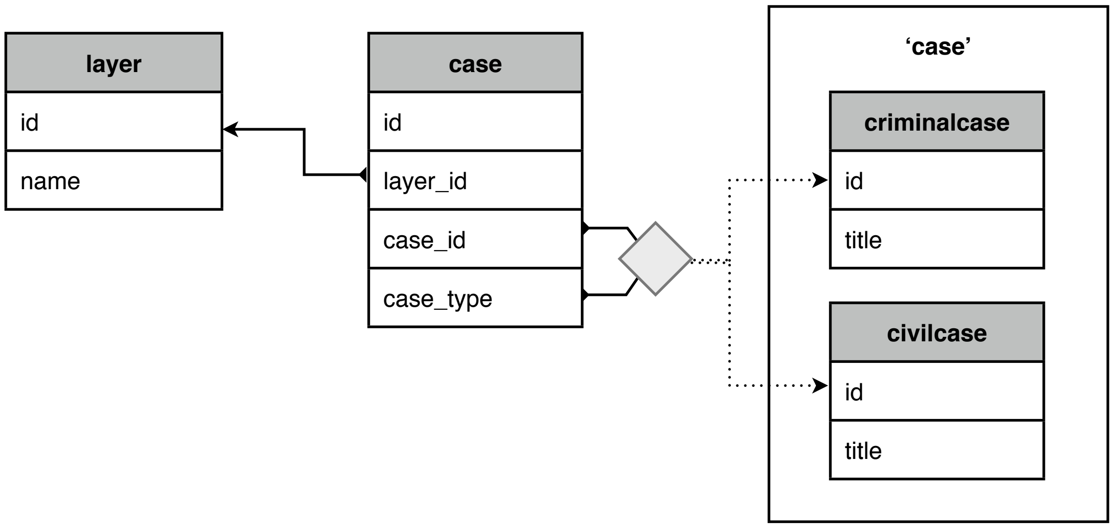
Accessing the Related Data
Because of the in-between ‘item’ table, we will need to perform a couple of queries in order to find related objects. The first query will be to find the related object in the ‘item’ table, and the second to find the objects related to the ‘item’ object in the related table:

```
<?php
	// accessing the borrower of a certain dvd or book:
	// first, find the ‘item’:
	$values[‘id’] = $dvd->getID(); // could just as well be a book
	$values[‘type’] = $dvd->getMeta(‘type’);
	$item = R::findOne(‘item’,
	                    ‘ item_type = :id AND item_id = :type’,
	                    $values);
	// now, find the borrower for that ‘item’:
	$values[‘id’] = $item->borrower_id;
	$values[‘type’] = $item->borrower_type;
	$borrower = R::findOne(   $item->borrower_type,
	                          ‘ type = :type AND id = :id’,
	                          $values);

	// accessing an array of books and dvds borrowed by a given borrower:
	// first, find all ‘items’ borrowed by this borrower:
	$values[‘id’] = $student->getID(); // could just as well be a ‘teacher’
	$values[‘type’] = $student-getMeta(‘type’);
	$borrowed = R::find(‘item’, ‘ type = :type AND id = :id’, $values);
	// now, use this array to fetch all related books and dvds:
	foreach($borrowed as $item) {
		$key = $item->getID();
		$items[$key] = R::find(   $item->item_type,
		                          ‘ id = ?’,
		                          array($item->item_id));
	}
	//$items will have all books and dvds borrowed by the related borrower
?>
```

Many-to-PolyMany

Definition

A *many-to-polymany* relationship is similar to a simple many-to-many relationship, but here one of the 'many' objects is of a fixed type, while the type of the other can vary. The second 'many' object in the relationship is therefore said to be 'polymorphic'. The database tables of a many-to-polymany relationship usually look like this:



This system allocates layers to the legal cases in which they are working. A layer may be allocated to several cases, and a case may be handled by several layers at once. This characterises a many-to-many relationship between layers and cases. There are, however, different types of 'case' objects: *criminalcase* and *civilcase*. As one of the 'many' objects can be of different types, this is a polymorphic relatioship on one side - a many-to-polymany.

An in-between 'case' table is necessary to keep the relationship between cases and their lawyers, so we can track not just the related case id, but also the case type.

Redbean Example

We are setting up a system to record information about actors, and about the various films and plays in which they performed in their career. Each actor may have had several roles, having performed in many plays and films. Each play or film, on the other hand, will have a list of several actors in the cast. This establishes a many-to-many relationship between actors and their films and plays - that is, their roles. As 'roles' represent different types of objects - a film or a play - this will be a polymorphic relationship on one side.

Creating the Objects

Using *R::dispense()*, create *actor, film* and *play* objects, populating them with properties as required. Use *R::store()* to save the objects to the database. Redbean will create the 'actor', 'film' and 'play' tables automatically for you, if they don't already exist.

```
<?php
      // creating an ‘actor’ object, to be stored in a table named ‘actor’:
      $actor= R::dispense(‘actor’);
      $actor->name = “John Smith”; // add properties as needed

      // store object to save in the database:
      R::store($actor);

      //...repeat as above for ‘film’ and ‘play’ objects.
?>
```

Establishing the Relationship
In our example, we will have 3 actors: actor1, actor2 and actor3. Actor1 and actor2 have acted together in a film, while actor2 and actor3 have acted together in a play. Once all actor, film and play objects are created, we can establish the relationships.

We do not, however, wanto to link our ‘actor’ table directly to the film and play tables. Rather, we want to have a generic ‘role’ table in between, and establish the relationship between actors, films and plays through that - view diagram at the beginning of chapter. Establishing the connections between actors and films/plays, therefore, will be a 2-stage process: first, we must create a generic ‘role’ object, and then we link that role to both an actor, and then to a film or play:

```
<?php
      // use R::dispense() to create ‘role’ objects, and a ‘role’ table:
      list($role1, $role2, $role3, $role4) = R::dispense(‘role’, 4);

      // link a film or play to a role by storing its id+type:
      $role1->role_id = $film->getIO();
      $role1->role_type = $film->getMeta(‘type’);
      // we can then link an actor to a role
      // by nesting the actor inside the role:
      $role1->actor = $actor1;

      $role4->role_id = $play->getIO();
      $role4->role_type = $play->getMeta(‘type’);
      // we can also link an actor to a role
      // by directly storing the actor id in the role:
      $role4->actor_id = $actor3->getID();

      // store ‘role’ objects to save relationships in database:
      R::store($role1); R::store($role4);

      $role2->role_id = $film->getIO();
      $role2->role_type = $film->getMeta(‘type’);
      $role3->role_id = $play->getIO();
      $role3->role_type = $play->getMeta(‘type’);
      // we can also link a list of roles to an actor at once,
      // using the magic sharedType property:
      $actor2->sharedRole = array($role2, $role3);

      // store ‘actor’ object to save relationships in database:
      R::store($actor2);
?>
```

Accessing the Related Data

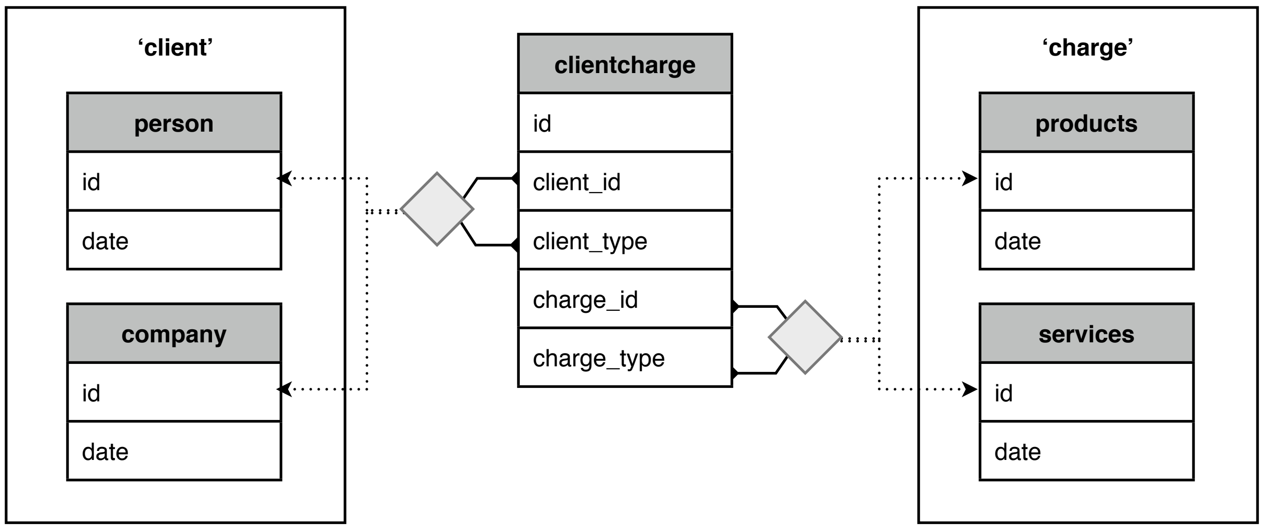
Because of the in-between ‘role’ table, we will need to perform a couple of queries in order to find related objects. The first query will find the related objects in the ‘role’ table, and the second will find the objects related to the ‘role’ objects in the related ‘many’ table:

```
<?php
      // accessing an array of films and plays given a certain actor:
      // we’ll build an array replacing every role in the magic ‘ownRole’
      // with its corresponding film or play:
      foreach($actor->ownRole as $role) {
            $key = $role->getID();
            $roles[$key] = R::findOne( $role->role_type,
                                    ‘ id = ?’,
                                    array($role->role_id));
      }
      //$roles will have all films and plays related to the given actor

      // accessing an array of related actors given a certain film or play:
      // first, find the ‘roles’ for the given film or play:
      $values[‘id’] = $film->getID(); // could just as well be a play
      $values[‘type’] = $film->getMeta(‘type’);
      $roles = R::find(   ‘role’,
                          ‘ role_type = :type AND role_id = :id’,
                          $values);
      // now, use this array to fetch all related actors:
      foreach($roles as $role) {
            $actor = R::relatedOne($role, ‘actor’);
            $key = $actor->getID();
            $actors[$key] = $actor;
      }
      //$actors will have all actors who acted in the related film or play
?>
```

PolyMany-to-PolyMany

Definition

A *polymany-to-polymany* relationship is similar to a simple many-to-many relationship, but here the type of the objects on both sides of the relationship is not fixed, and can vary. The objects on both sides of the relationship are therefore said to be 'polymorphic'. The database tables of a polymany-to-polymany relationship can typically look like this:



This system keeps track of charges incurred by clients. We have 2 types of clients: *persons* and *companies*. We supply these clients with *products*, as well as with *services,* for which we charge. A client, therefore, can have several charges for products and services. Each charge, however, will be allocated to a single client, which will be a person or a company. This characterises a one-to-many relationship between clients and charges. Both clients and charges, however, can be different types of objects: clients can be *persons* or *companies*, and charges can be *products* or *services*. This makes it a polymorphic relationship on both the 'one' and the 'many' sides.

An in-between 'item' table is necessary to keep the relationship between client and charge, and it has to store the ids and types for both the client and the charge.

Redbean Example

We are setting up a project management system for a school, in which we assign projects to different staff members. A staff member can be part of several projects at once, and each project can be handled by several staff members. This establishes a many-to-many relationship: many staff to many projects.

The staff, however, will sometimes be internal *employees*, and sometimes external *contractors* - ie., a staff member may be different types of objects. Projects, on the other hand, can also be of different types. The school sets up projects for taking student group on travel *tours*, and also sets up projects for developing new *courses*. This means that the objects on both sides of this relationship can be different types of objects, and are polymorphic. This is a polymany-to-polymany relationship.

Creating the Objects
Using *R::dispense()*, create *employee, contractor, tour* and *course* objects, populating them with properties as required. Use *R::store()* to save the objects to the database. Redbean will create the 'employee', 'contractor', 'tour' and 'course' tables automatically for you, if they don't already exist.

```
<?php
      // creating an ‘employee’ object,
      // to be stored in a table named ‘employee’:
      $employee= R::dispense(‘employee’);
      $employee->name = “John Smith”; // add properties as needed

      // store object to save in the database:
      R::store($employee);

      //...repeat as above for ‘contractor’, ‘tour’ and ‘course’ objects.
?>
```

Establishing the Relationship
Once employee, contractor, tour and course objects have been created, we can establish relationships between them. Imagine we have a couple of new projects - a new course and a new tour - and we want to assign several staff members to these projects. Some of the staff working in the projects will be employees, but some will be external contractors.

To establish the relationships between a staff member and a project, we will create a special 'staffproject' object. Because both our 'staff' and 'project' are polymorphic, the 'staffproject' object will have to keep track not just of the id of each object in the relationship, but also of its type. We can use the *getID()* method to find out an object's id, and the *getMeta('type')* method to find out its type. We can then explicitly store the relationship information in the database, simply by storing the 'staffproject' - this will create the in-between polymorphic table, and establish the relationships for us:

```
<?php
      // creating a ‘staffproject’ object,
      // to be stored in a table named ‘staffproject’:
      $staffproject= R::dispense(‘staffproject’);

      // linking a project to it, by storing its id+type:
      $staffproject->project_id = $tour->getID() // could also be a $course
      $staffproject->project_type = $tour->getMeta(‘type’);

      // linking a staff member to it, by storing its id+type:
      $staffproject->staff_id = $employee->getID(); // could also be $contractor
      $staffproject->staff_type = $employee->getMeta(‘type’);

      // store ‘staffproject’ object to save the relationship in the database:
      R::store($staffproject);

      //...repeat as above to link as many staff to as many projects as needed.
?>
```

Accessing the Related Data

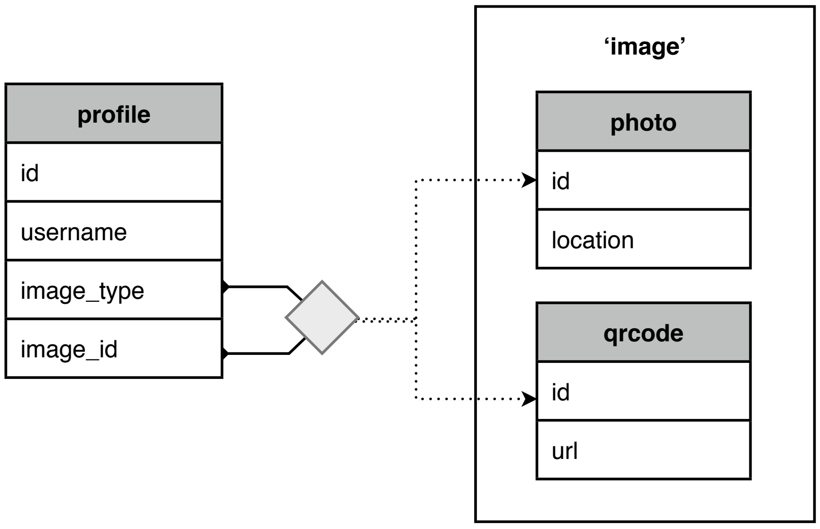
Because of the polymorphic objects on both sides of the relationship, we will need to perform a couple of queries in order to retrieve the related objects on the other side of the relationship. The first query will gives us the records in the in-between ‘staffproject’ table, and the second query will give us the records related to the ‘staffproject’ records in the related ‘many’ table:

```
<?php
	// accessing an array of staff associated with a given project:
	// first, find all ‘staffproject’ records in the in-between table -
	// note that $project could be a ‘tour’ or a ‘course’ object:
	$values[‘type’] = $project->getMeta(‘type’);
	$values[‘id’] = $project->getID()
	$staffprojects = R::find( ‘staffproject’,
	                          ‘ project_id = :id AND project_type = :type’,
	                          $values);
	// now, use this array to fetch all related staff:
	foreach ($staffprojects as $member) {
		$type = $member->staff_type;
		$id = $member->staff_id;
		$key = $member->getID();
		$staff[$key] = findOne( $type, ‘ id = ?’, array($id));
	}
	// $staff will have all ‘employee’ and ‘contractor’ objects
	// related to the given ‘tour’ or ‘course’

	// accessing an array of projects associated with a given staff member:
	// first, find all ‘staffproject’ records in the in-between table -
	// note that $staff could be an ‘employee’ or a ‘contractor’ object:
	$values[‘type’] = $staff->getMeta(‘type’);
	$values[‘id’] = $staff->getID()
	$staffprojects = R::find( ‘staffproject’,
	                          ‘ staff_id = :id AND staff_type = :type’,
	                          $values);
	// now, use this array to fetch all related projects:
	foreach ($staffprojects as $proj) {
		$type = $proj->project_type;
		$id = $proj->project_id;
		$key = $proj->getID();
		$projects[$key] = findOne( $type, ‘ id = ?’, array($id));
	}
	// $projects will have all ‘tour’ and ‘course’ objects
	// related to the given ‘employee’ or ‘contractor’
?>
```

One-to-PolyOne

Definition

A *one-to-polyone* relationship is similar to a simple one-to-one relationship, but here the type of the object on one side of the relationship can vary, while on the other side it is fixed. The object on one side of the relationship is therefore said to be 'polymorphic'. The database tables of a one-to-polyone relationship may look like this:



This system allows users to attach an image to their profile. An image may be a *photo* uploaded by the user, or a QR code image which is automatically built into the system. Each image (photo or QR code) may only be connected to a single profile, and each profile may only have one representative image. This established a one-to-one relationship: one image to one profile, and one profile to one image. The images, however, can be different types of objects (photos or QR codes). This makes it a polymorphic relationship on one side: a one-to-polyone.

As ca be seen here, the table setup for this type of relationship is similar to a simple one-to-many, but an extra 'image_type' field has to be used to help select the table to which the image belongs.

Redbean Example

We have a system where a user's payment method is stored within the system. The user may choose to pay with their paypal account, or by direct debit from their bank account. We will therefore have 'user', 'paypal' and 'bankaccount' objects.

Each payment method may be connected to just a single user - that is, a bank or paypal account can only be linked to one user. On the other hand, each user may pay using only one type of payment. This establishes a one-to-one relationship: one payment method for one user. In this case, however, one of the sides - the 'payment' side - can be more than one type of object (a paypal or a bank account). This is a one-to-polyone relationship.

Creating the Objects
Using *R::dispense()*, create *user, paypal* and *bankaccount* objects, populating them with properties as required. Use *R::store()* to save the objects to the database. Redbean will create the 'user', 'paypal' and 'bankaccount' tables automatically for you, if they don't already exist.

```
<?php
	// creating a ‘user’ object, to be stored in a table named ‘user’:
	$user= R::dispense(‘user’);
	$user->name = “John Smith”; // add properties as needed

	// store object to save in the database:
	R::store($user);

	//...repeat as above for ‘paypal’ and ‘bankaccount’ objects.
?>
```

Establishing the Relationship
Once users, paypal and bankaccounts are created, we can establish connections between them. In this example, let’s say our system has 2 users, one paying by paypal, the other with a bank account.

Because the payment method is polymorphic, in the ‘user’ object, we will need to store not just the the payment method’s id, but also its type:

```
<?php
	// create a link by storing the paymethod id+type:
	// note: the paymethod could be either a ‘paypal’ or ‘bankaccount’ object:
	$user1->paymethod = $paymethod->getID();
	$user1->paymethod_type = $paymethod->getMeta(‘type’);

	//...repeat for $user2

	// store ‘user’ objects to save relationships in the database:
	R::store($user1); R::store($user2);
?>
```

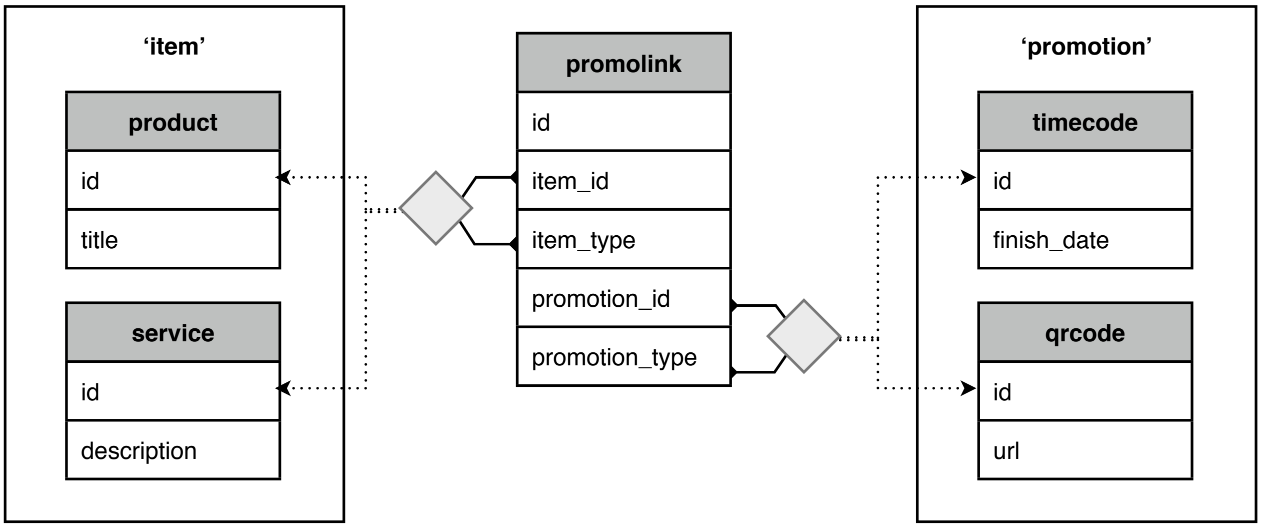
Accessing the Related Data
We need to perform a very simple query in order to obtain the payment method related to a user, or to find the user of any given payment method:

```
<?php
	// accessing related paymethod given a user:
	$type = $user->paymethod_type;
	$id = $user->paymethod_id;
	$paymethod = R::findOne($type, ‘ id = ?’, array($id));

	// accessing related user given a paymethod -
	// note that paymethod can be either ‘bankaccount’ or ‘paypal’:
	$type = $paymethod->getMeta(‘type’);
	$id = $paymethod->getID();
	$user = R::findOne(‘user’,
				‘ paymethod_type = :type AND paymethod_id = :id’,
				array($type, $id));
?>
```

PolyOne-to-PolyOne

Definition

A *polyone-to-polyone* relationship is similar to a simple one-to-one relationship, but here the type of the objects on both sides of the relationship is not fixed, and can vary. The objects on both sides of the relationship are therefore said to be 'polymorphic'. The database tables of a polyone-to-polyone relationship might look like this:



This system allows the user to attach promotions to items sold by the company. Each item can only have 1 running promotion, and each promotion can only be attached to 1 item. We have, therefore, a one-to-one relationship: one item to one promotion. Both items and promotions, however, can be different types of objects. Items can be *products* or *services*, and promotions are run differently based on whether they are *timecode*, or *qrcode* promotions. This means we have polymorphic objects on both sides of the relationship, and this is a polyone-to-polyone relationship.

An in-between 'promolink' table is necessary to keep the relationship between item and promotion, and it has to store the ids and types for both the item and the promotion.

Redbean Example

We are setting up a system in the website of our local school to allow students and staff to add profile information about themselves. In their profiles, users will be able to add a photo, if they wish. If the user does not have a photo, the system will automatically generate a QR code image that will be displayed instead - which might have some information about the user, or link to the user's profile page. Each profile can only have one image, and each image can belong to just one profile. This establishes a one-to-one relationship between profiles and images.

Both images and profiles, however, can be different types of objects: images can be *photos* or *qrcodes*, and profiles can represent *students* or *staff*. This makes it a polymorphic relationship on both sides: a *polyone-to-polyone* relationship.

Creating the Objects
Using *R::dispense()*, create *student, staff, photo* and *qrcode* objects, populating them with properties as required. Use *R::store()* to save the objects to the database. Redbean will create the ‘student’, ‘staff’, ‘photo’ and ‘qrcode’ tables automatically for you, if they don’t already exist.

```
<?php
      // creating a ‘student’ object, to be stored in a table named ‘student’:
      $student= R::dispense(‘student’);
      $student->name = “John Smith”; // add properties as needed

      // store object to save in the database:
      R::store($student);

      //...repeat as above for ‘staff’, ‘photo’ and ‘qrcode’ objects.
?>
```

Establishing the Relationship
Once student, staff, photo and qrcode objects are created, we can establish links between them. To establish the relationships between a profile and its image, we will create a special ‘profileimage’ object. Because both our ‘profile’ and ‘image’ are polymorphic, the ‘profileimage’ object will have to keep track not just of the id of each object in the relationship, but also of its type. We can use the *getID()* method to find out an object’s id, and the *getMeta(‘type’)* method to find out its type. We can then explicitly store the relationship information in the database, simply by storing the ‘profileimage’ - this will create the in-between polymorphic table, and establish the relationships for us:

```
<?php
      // creating a ‘profileimage’ object,
      // to be stored in a table named ‘profileimage’:
      $profileimage= R::dispense(‘profileimage’);

      // linking an image to it, by storing its id+type -
      // note that $image can be either a ‘photo’ or a ‘qrcode’ object:
      $profileimage->image_id = $image->getID()
      $profileimage->image_type = $image->getMeta(‘type’);

      // linking a profile to it, by storing its id+type -
      // note that $profile can be either a ‘student’ or a ‘staff’ object:
      $profileimage->profile_id = $profile->getID();
      $profileimage->profile_type = $profile->getMeta(‘type’);

      // store ‘profileimage’ object to save the relationship in the database:
      R::store($profileimage);

      //...repeat as above to link as many profiles to as many images as needed.
?>
```

Accessing the Related Data
We need to perform a very simple query in order to obtain the image related to a certain profile, or to find the profile connected to a given image:

```
<?php
      // accessing related image given a profile -
      // note that $profile can be either a ‘student’ or ‘staff’ object,
      // and that the resulting $image can be a ‘photo’ or ‘qrcode’ object:
      // first, we find the ‘profileimage’ object in the in-between table:
      $type = $profile->getMeta(‘type’);
      $id = $profile->getID();
      $profileimage = R::findOne(‘profileimage’,
                        ‘ profile_type = :type AND profile_id = :id’,
                        array($type, $id));
      // then, we find the related image:
      $type = $profileimage->image_type;
      $id = $profileimage->image_id;
      $image = R::findOne($type, ‘ id = ?’, array($id));

      // accessing related profile given an image -
      // note that $image can be either a ‘photo’ or ‘qrcode’ object,
      // and that the resulting $profile can be a ‘student’ or ‘staff’ object:
      // first, we find the ‘profileimage’ object in the in-between table:
      $type = $image->getMeta(‘type’);
      $id = $image->getID();
      $profileimage = R::findOne(‘profileimage’,
                        ‘ image_type = :type AND image_id = :id’,
                        array($type, $id));
      // then, we find the related profile:
      $type = $profileimage->profile_type;
      $id = $profileimage->profile_id;
      $profile = R::findOne($type, ‘ id = ?’, array($id));
?>
```

Beyond Basic

Apart from the basic relationship types mentioned in this guide, there are many other, more complex, variations. Hopefully, if you understand how to use RedbeanPHP to setup basic relationships appropriately, when the need arises for more complex solutions, you will have the skills to implement them confidently.

Some of the several more complex relationship types not covered in this guide include:

Polymorphic Self-Referential

Imagine you are setting up an online store, where in the catalogue page for a certain product, you will have a section for ‘people who bought this also bought...’ other products. This is a self-referential many-to-many relationship - one product can be listed on many other products’ pages, and vice-versa. Each product, however, can be different types of objects, with different data and functional requirements. We might have a store that sells books, dvds, software, electrical appliances, and even clothing. As each ‘product’ can actually be a different type of object, this relationship is also polymorphic.

There are 3 types of polymorphic self-referential relationships: one-to-one, one-to-many and many-to-many. Can you visualise the diagrams for these relationships, and how to implement them using RedbeanPHP?

Polymorphic with Variable Cardinality

Let us say we are setting up ‘user profiles’ for our website, and we decide that each user is going to be allowed to upload a photo that can be used throughout the site to represent the user in posts and discussions. If the user does not have a photo that they want to upload, we will allow them to select from a list of previously uploaded icons that we’ll have available in the system.

At first, this may seem to be a one-to-polyone relationship: one profile for each ‘image’, where the ‘image’ is either an uploaded photo, or a built-in icon. The problem here is that this relationship is not one-to-one in the opposite direction: while each uploaded photo will only be connected to one profile, each built-in icon in the system may be in use by several users at once. So, if the polymorphic object is a *photo*, then the relationship is one-to-one, but if the object is an *icon*, then the relationship is one-to-many. The relationship will vary its **cardinality** depending on the polymorphic object. We can call this a ‘polymorphic with variable cardinality’ - or *polycardinal* - relationship.

‘Through’ Relationships

As your system becomes more complex, and your data diagram expands, objects start to have ‘indirect’ relationships with each other, through shared links with other objects. Imagine, for instance, that you have a system for invoices, and payments. The relationship between invoices and payments is a simple many-to-many: once invoice can have several payments attached, and one payment may be paying for several invoices at once. Each invoice, however, is also linked to several ‘charges’ - one charge for each product or service listed in the invoice.

There is no direct connection between payments and individual charges - payments are connected to invoices, and invoices are connected to charges. But we can say that *payments* are linked to *charges* **through** *invoices*.