# Pclib ORM

## Úvod

ORM je vrstva pro práci s databází pomocí objektového přístupu. V architektuře MVC patří do "M" (model). Zatímco v jiných frameworcích má implementace ORM desítky až stovky tříd, v pclib si vystačíme se dvěma.

```
require 'pclib/pclib.php';

use pclib\orm\Selection;
use pclib\orm\Model;

$app = new PCApp('test');
$app->db = new PCDb('mysql://localhost/test/utf8');
```

## Selection a Model

### Selection

Třída **Selection** je vlastně dotaz do databáze obalený objektem. Nebo si můžeme jako analogii představit třeba selektory v jquery.

```
$sel = new Selection;
$sel->from('books')->where('price<200')->order('title')->limit(100);
```

Používá tzv. *fluent interface*, to znamená, že volání jednotlivých metod je možné libovolně řetězit za sebe. Výsledkem je databázový dotaz.

```
$sel->getSql();
```

zobrazí:

```
select * from books where price<200 order by title limit 100
```

Je možné přidávat další podmínky i dodatečně:

```
if ($filter == 'hobby') {
      $sel->where("category='hobby'");
}

//select * from books where price<200 and category='hobby' order by title limit
100
```

Dokud se nepoužije nějaká exekutivní metoda, Selection k databázi nijak nepřistupuje, ale pouze si ukládá parametry dotazu.

Lze vrátit výsledek jako pole:

```
$sel->toArray()
```

Vrátit první záznam:

```
$sel->first();
```

Počet záznamů:

```
$sel->count();
```

Smazat celý výběr:

```
$sel->delete();
```

Aktualizovat výběr:

```
$sel->update(['price' => 300]);
```

a další funkce.

Je možné procházet výběr pomocí foreach:

```
foreach($sel as $model) {
      print $model;
}
```

Selection v tomto případě vrací záznamy po jednom (fetch), takže můžeme mít miliony záznamů a nedojde k tomu, že by se interně načítaly do nějakého pole, které by zabralo celou paměť.

Lze jej také použít v gridu místo dotazu:

```
$grid = new pclib\Grid;
$grid->setSelection($sel);
print $grid;
```

Poznámky:

- Pro filtrování podle vazební tabulky použijte funkci *whereJoin()*. Další funkce viz referenční manuál. class Selection
- V metodě controlleru můžete vytvořit selection také zkratkou:
  *$sel = $this->selection('books')->where...*
- Lze používat parametry: *->where("id='{0}'", $id)*

## Model

ORM znamená Objektově relační mapování [wiki], jedná se tedy o jakési objekty. Selection proto implicitně vrací pro každý záznam z databáze objekt **Model**.

Třída Model je inspirována návrhovým vzorem ActiveRecord [wiki], takže například pole z databáze jsou proměnné objektu.

```
$book = $sel->first();
print $book->title;
```

Změna titulku knihy:

```
$book->title = 'Nový titulek';
$book->save();
```

(aktualizují se jen pole, která byla doopravdy změněná - update bude obsahovat pouze sloupec 'title')

Smazání knihy z databáze:

```
$book->delete();
```

Vrácení záznamu v podobě pole:

```
$book->toArray();
```

Načtení knihy podle primárního klíče:

```
$book->find($id);
```

($book nyní obsahuje knihu s příslušným id)

Vytvoření nového záznamu v databázi z pole values:

```
$values = ['title' => 'Nová kniha', ...];
$book = new Model('books', $values);
$book->save();
```

V Controlleru je možné získat model podle primárního klíče i zkratkou pomocí metody *->model()*. Tj. uvnitř akce Controlleru:

```
$book = $this->model('books', $id);
```

Jedna hezká věc na modelech je automatické joinování tabulek. Velice to zpřehledňuje a zjednodušuje kód, pro často používané situace. Dejme tomu, že máme tabulku **authors**, která je s tabulkou **books** spojená vazbou 1:N, tj. každý autor může mít n-knih a spojení je pomocí pole **books.author_id**.

Pak si můžeme jednoduše vrátit autora knihy:

```
$author = $book->author;
```

Nebo rovnou jeho jméno:

```
$author = $book->author->name;
```

Jde to i obráceně:

```
$author->books
```

Vrátí všechny knihy autora. A protože vrácený objekt je Selection, můžeme na výsledku používat všechny jeho metody. Například seřadit knihy autora podle abecedy a vypsat:

```
foreach($author->books->order('title') as $book) {
      print $book->title;
}
```

Autojoiny za vás obstarají i kaskádní dotazy. Na to se často zapomíná, pokud v průběhu času v aplikaci přibývají tabulky a mazáním zůstávají v databázi například sirotci bez rodičů.

```
$author->delete()
```

Se zachová, podle nastavení, dvěma způsoby:
1) Vyhodí výjjimku, pokud existují nějaké vazební záznamy (knihy autora)
2) Vymaže autora včetně všech vazebních záznamů (zde knih v tabulce books)

Jak ORM ví, jakým způsobem joiny vytvořit?
Jednoduše. :) Přečte si to v šabloně modelu, kterou vytvoříme v adresáři **models/templates/**

V **books.tpl** bude řádek **relation**:

```
<?elements
class model table "books"
relation author table "authors" key "author_id" owner
?>
```

V **authors.tpl**:

```
<?elements
class model table "authors"
relation books table "books" key "author_id" many
?>
```

Šablony mohou obsahovat i jiné konfigurační údaje, například validační podmínky (stejné, jako pro formulář).

```
<?elements
class model table "books"
column title required
?>
```

Další metody třídy **Model** viz referenční manuál.[class Model]

# Architektura MVC

Možná si říkáte, že je zbytečné vytvářet pro každý řádek tabulky objekt. Ve skutečnosti je to docela silný koncept. Ještě jsme neřekli, že si můžete vytvořit vlastní třídy modelů.
Stačí v adresáři **models/** vytvořit soubor např. **BooksModel.php** s třídou modelu:

```
class BooksModel extends PCModel {
	...
}
```

Selection i ostatní metody pak místo třídy Model budou pro tabulku books vracet vaši třídu **BooksModel**.
A do ní můžeme soustředit veškeré funkce, které se logicky vztahují na entitu "kniha".
např.

```
$book->cenaSDph();
$book->objednat();
$book->kusuNaSklade();
```

V šabloně je možné definovat i validační pravidla. Např. povinnost vyplnění názvu knihy a autora. A tato validační pravidla budou platit všude tam, kde se objekt **$book** objeví. To je výhoda oproti obvyklé validaci vázané na formulář, protože formulářů a vkládání knih může být několik - např. v administraci, při objednávání, při importu knih z externího systému apod. A jak všichni víme DRY (don't repeat yourself) je lepší než WET (we enjoy typing). [wiki]

V MVC je obvykle doporučovaný postup mít tzv. tenký controller a veškerou aplikační logiku soustředit do vrstvy modelu. (zatímco vše, související se zobrazováním, je ve vrstvě pohledu - view).
Controller pouze na základě uživatelského vstupu (kliknutí na odkaz), rozhodne jakou akci aplikace zavolat.

```
function showAction($id)
{
	$book = $this->model('books', $id);
	//metoda, kterou predame book do zobrazovaci sablony
	$view = $this->view('book', $book);
	return $view->html();
}
```

Výhoda je, že pokud potřebujeme vyexportovat stránku s nabídkou knihy třeba do pdf, do xml, do editace v adminu, do zkráceného zobrazení v košíku... není nic jednoduššího:

```
function exportPdfAction($id)
{
	$book = $this->model('books', $id);
	$view = $this->view('book_pdf', $book);
	return $view->html();
}
```

Všude, kde se entita **book** objeví, máme automaticky k dispozici veškerou logiku s ní svázanou a nemusíme vymýšlet, kam a do kterého include souboru ji umístit.
A konečně tím, že máme aplikační logiku soustředěnou v samostatné izolované jednotce, je možné vytvářet automatizované jednotkové testy.

```
class BooksModelTest extends PHPUnit_Framework_TestCase {
	test() {
```

```
            $book = new BooksModel;
            $book->find(1);
            $this->assertTrue($book->isInDb());
            $this->assertEquals($book->cenaSDph(), 110);
            ...
        }
}
```