Agile Data

Release 1.2-latest

Jul 23, 2019

Contents

Contents:

CHAPTER 1

Quickstart

The reason most developers dislike *Object Relational Mapper (ORM)* frameworks is because it is slow, clumsy, limited and flawed. The benefits are **consistency, compatibility and abstraction** it offers to larger projects.

ATK Data is a Data/Persistence mapping framework in PHP implementing **an alternative to ORM** to achieve the following goals:

- offers consistency, compatibility and abstraction
- avoid classical flaws of ORM pattern
- remain SQL/NoSQL agnostic
- offer ability to take advantage of vendor-specific features

The pattern implemented by ATK Data (DataSets) were proposed on *Apr 2016*. The design goals and concepts are further discussed in *Overview*.

1.1 Quick Intro

ATK Data is focused on reducing number of database queries, moving CPU-intensive tasks into your database (if possible). It is well suited for Amazon RDS, Google Cloud SQL and ClearDB but thanks to abstraction will work transparently with Static data, NoSQL or RestAPIs backends.

When using ATK Data, it's possible to work with the data on a **higher level**, performing operations such as *DeepCopy*, *Deep Traversal* as well as Aggregation.

Core of ATK Data aims to:

- allow you to describe your object entities without database specifics
- save/load model data through generic database drivers for SQL and NoSQL
- work with multiple record sets in your application
- integrate with generic User Interface or Application Interface extensions

ATK Data can do a lot more through add-ons.

1.2 Quick Links

Model Definition: Defined in PHP class method init(). Supports PDO, NoSQL, Array, Session, CSV, and RestAPI through custom persistence classes. Column-specific structure but support nesting and containment. Inheritance.

Loading and Storing data: Generic integration with multiple SQL vendors. Support for Expressions. Table name mapping, field name mapping, type mapping, serialization, field-level encryption. Sub-Selects. Joins. Mapping to stored procedures. Recursive import. Hooks. Behaviours.

DataSet Operations: Additive conditions. Expression-based conditions. Limits. Fetching all data, selective columns or mapping. Expressing and re-using SQL statements. Updating multiple records. Aggregation functions. Inferring values. Global conditions.

Field Types: Native PHP types. Custom types. Typecasting settings (e.g. format). ENUMs. Key-value. PHP Calculated types.

References and Relations: hasOne and hasMany reference. Traversing without data loading. Cross-persistence traversal. Deep traversal.

Utilities: Schema migration. Deep copy. Unions. Aggregate Models.

Actions: Defining user actions. Action arguments. Action executor specs. ACL. Transactions.

Meta Information: Inferring field decorator. Validation rules. Captions, hints and description. Localization.

Security: Scopes. System fields and actions. Areas of concern.

Refactoring: Using with existing schema. Refactoring database.

1.3 Quick Tour

First get the following:

- PHP 7.0 or above.
- MySQL or MariaDB
- Install Agile Data Primer

```
git clone https://github.com/atk4/data-primer.git
cd data-primer
composer update
cp config-example.php config.php

# EDIT FILE CONFIG.PHP
vim config.hpp

php console.php
```

Enter statements into console one-by-one and carefully observe results. If you wish to see SQL queries as they are being executed, be sure to include "dumper" proxy.

1.3.1 Persistence Driver

Persistence is a database, like MySQL. It could also be a CSV file. To interract with a persistence you need a driver. *console.php* has already initialized persistence and connected to database, but no queries were executed:

```
> $db
=> atk4\data\Persistence\SQL {...}
```

The appropriate persistence class will be selected depending on your connection string (DSN).

1.3.2 Model Definition

Your application does not talk to database directly. Instead it requires an object, which we call *Model*. You should create class for every business entity, for example:

- Client
- Invoice
- InvoiceLine

1.3.3 Model Instance

Return back to the console, and create instance of Client class:

```
> $client = new inv\Client($db);
=> inv\Client {#170
     +id: null,
     +conditions: [],
   }
```

The object *$client* has a state and can be used to interact with single or multiple records. Multi-record operations currently apply to entire set of data. Lets find out how many cliens we have:

```
> $client->action('count')->getOne();
=> "10"
```

Next, we can use `Model::loadAny` to load one record from persistence and then get data with `Model::get`:

```
> $client->loadAny();
> $client->get();
```

The types returned by *get()* are automatically converted from database-specific to PHP-specific, such as *DateTime*.

1.3.4 Fields

Model object will also populate `Field` objects. You can get list of them with `Model::getFields`. Observe that field objects may vary depending on definition or Data Persistence capabilities.

Unlike other frameworks, Model object is reusable. You can unload and load data of another record or even iterate through entire set:

```
> $client->unload()
> $client->loadBy('name', 'John');
```

Field objects also remain and can hold valuable information which may be relied on by other frameworks or add-ons on the fly.

1.3.5 References

ATK Data uses term "reference" instead of "relation", because it's more broad. Think of it this way:

- one record of Client has many Invoice records.

Reference is defined in `Model::init` method like this:

```
$this->hasMany('Invoices', Invoice::class);
```

Go back to console and see which references your *$client* object has:

```
> $client->getRefs();
```

Then traverse this reference:

```
> $invoices = $client->ref('Invoices');
=> inv\Invoice {#226
     +id: null,
     +conditions: [
       [
         "inv_client_id",
         "45",
       ],
     ],
   }
```

Observe that the model returned by `Model::ref` does not have active record, but instead it has condition set. This narrows down set of "All invoices" to the "Invoices of client John". We can execute operation on John's invoices:

```
> $invoices->action('count')->getOne();
=> "2"
```

You do not have to load record in order to traverse further. Try this:

```
> $all_lines = $invoices->ref('Lines');
```

You will get a Line object conditioned to a DataSet corresponding to all invoices of client John. This time lets calculate total amount of all the invoice lines:

```
> $all_lines->action('fx', ['sum', 'total']);
=> "69"
```

The query used to fetch this value was constructed with our inferred conditions, but also taking into account that there are no physical "total" field and instead it is a multiplication of *qty* and *price* fields.

Our invoices also have a *due* field, lets see how many invoices are due:

```
> $due = clone $invoices;
> $due->addCondition('due', '>', 0);
> $due->export(['ref', 'total', 'due']);
```

This would give you list of due invocies and amount due.

1.3.6 User Actions

ATK Data provides a way to describe User actions. Once described action can be invoked through generic API, Add-on or UI. Lets find out which user actions *$invoices* offers:

```
> $invoices->getActions()
```

You should see action *register_payment* here as well as description of it's arguments. Lets invoke this action:

```
> $invoices->register_payment(30.0);
```

Now you can re-request list of due invoices:

```
> $due->export(['ref', 'total', 'due']);
```

This time you should see a different picture, since the payment was allocated towards multiple invoices of client 'John'.

1.4 Quick UI

ATK UI contains enough information about your business model to actually be able to create a very nice administration system for it. Not only that, but some elements can be used for the client-facing front-end too with minimum code.

1.4.1 Install Dependencies

ATK Data can be complimented by https://github.com/atk4/ui, which can be used in conjunction with any other meta-framework. Here I'll present just a quick intro focused on building UI for existing data structure, but for a more comprehensive intro, see https://agile-ui.readthedocs.io/en/latest/quickstart.html.

Use composer:

```
composer install atk4/ui
```

Next create a simple file:

```
$app = new \atk4\ui\App();
$app->dbConnect('mysql://root:root@localhost/atk');

// Specify which UI layout to use
$app->initLayout('Centered');

// Create new Form object
$form = $app->add('Form');

// Associate UI component with your model and persistence
$form->setModel(new Client($app->db));
```

Opening the page will display a form consistent with the model/field definitions. A generic UI component will find fields suitable for the form and present them accuratelly with a correct type. No extra files or code is required.

1.4.2 Try using different views

ATK UI comes with varietty of different views, so try replacing $form creation with this:

```
$table = $app->add('Table');
$client = new Client($app->db);

// Load existing client
```

(continues on next page)

(continued from previous page)

```
$client->load(1);

// Show invoices of specific client inside a table
$table->setModel($client->ref('Invoices'));
```

Next relace *Table* with *CRUD* and now your UI should allow you to add, edit and delete records too. Make note that any new invoices you add will be associated with the client with *id=1*:

```
$table = $app->add('Table');
$client = new Client($app->db);

// Load existing client
$client->load(1);

// Show invoices of specific client inside a table
$table->setModel($client->ref('Invoices'));
```

1.4.3 Use Admin layout

Finally - ATK UI offers a hierarchical approach to rendering UI, so you can easily design layouts:

```
$app = new \atk4\ui\App();
$app->dbConnect('mysql://root:root@localhost/atk');

// Admin layout offers menu for navigating
$app->initLayout('Admin');


// Load existing client
$client = new Client($app->db);
$client->load(1);

$columns = $app->add('Columns');

// Two column layout
$c_left = $columns->addColumn();
$c_right = $columns->addColumn();

// Show client card on the left and invoices on the right
$c_left->add('Card')->setModel($client);
$c_right->add('CRUD')->setModel($client->ref('Invoices'));
```

1.4.4 Don't forget to authenticate

I leave it as an exercise to you to create authentication for the admin. There is a very good add-on https://github.com/atk4/login which will make use of a Model to verify user access:

- require atk4/ui
- create 'User' model
- implement auth checking
- verify login/logout functionality
- verify password change screen

1.5 Quick API

If you need integration with React app or Mobile app, you might need an API. Once again - because ATK Data models contain some useful information already, it can be linked up with the API end-points directly. Also due to nature of https://github.com/atk4/api - it is a non-intrusive class, which follow standards and plays nice with other frameworks.

1.5.1 Install Dependency

Install using composer:

```
composer require atk4/api
```

1.5.2 Write the code

Create *api.php* file. You could mod_rewrite all requests into this file or use *api.php/clients/1* style endpoints, which would work out of the box:

```
$api = new \atk4\api\Api();

// Create end-point route for clients
$api->rest('/clients', new Client($db));

// Create end-point route for client invoices
$api->rest('/clients/:client_id/invoices', function($id) use($db) {
    $client = new Client($db);

    return $client->load($id)->ref('Invoices');
});
```

1.6 Actions, ACL and More

In a normal situation, your UI code may have to deal with various cases and variance depending on permissions, object state and more.

With ATK add-ons you can continue to focus your work on ATK Data models and simply have the UI / API reflect your structure and business rules.

So don't ask "how to add new button to the table" but rather thing in terms "how to add new action to a model". The benefit is that actions can also be accessed from the APIs if authentication and access control is configured correctly. You'll learn how to do that as you continue reading this documentation.

1.7 Conclusion

In ATK community there is a saying "way of ATK". This refers to an implementation which implements the requirement with very small amount of effort from developers.

This QuickStart presented only the basics and demonstrated inter-component integration. I recommend that as you continue to work on your models, keep "UI" and "API"

1.7.1 MasterCRUD Add-on

I simply have to mention MasterCRUD add-on (https://github.com/atk4/mastercrud), which is designed to simplify things even further. This add-on is ideal for Administration Systems and traversing relationships automatically. I leave it to you to investigate how your entire Admin System code could be even shorter.

CHAPTER 2

Overview

Business Model (see *Model*) You define business logic inside your own classes that extend `Model`. Each class you create represent one business entity.

Model has 3 major characteristic: Business Logic definition, DataSet mapping and Active Record.

See: `Model`

Persistence (see Data Persistence) Object representing a connection to database. Linking your Business Model to a persistence allows you to load/save individual records as well as execute multi-record operations (Actions)

For developer, persistence should be a secondary concern, after all it is possible to switch from one persistence to another and compensate for the feature differences without major refactoring.

DataSet (see DataSet) A set of physical records stored on your database server that correspond to the Business Model.

Active Record (see Active Record) Model can load individual record from DataSet, work with it and save it back into DataSet. While the record is loaded, we call it an Active Record.

Action (see Action) Operation that Model performs on all of DataSet records without loading them individually. Actions have 3 main purposes: data aggregation, referencing and multi-record operations.

2.1 Persistence Domain vs Business Domain

It is very important to understand that there are two "domains" when it comes to your data. If you have used ORM, ActiveRecord or QueryBuilders, you will be thinking in terms of "Persistence Domain". That means that you think in terms of "tables", "fields", "foreign keys" and "group by" operations.

In larger application developers does not necessarily have to know the details of your database structure. In fact - structure can often change and code that depend on specific field names or types can break.

More importantly, if you decide to store some data in different database either for caching (memcache), unique features (full-text search) or to handle large amounts of data (BigData) you suddenly have to carefully consider that in your application.

Business Domain is a layer that is designed to hide all the logic of data storage and focus on representing your business model in great detail. In other words - Business Logic is an API you and the rest of your developer team can use without concerning about data storage.

Agile Data has a rich set of features to define how Business Domain maps into Persistence Domain. It also allows you to perform most actions with only knowledge of Business Domain, keeping the rest of your application independent from your database choice, structure or patterns.

2.2 Class vs In-Line definition

Business model entity in Agile Data is represented through PHP object. While it is advisable to create each entity in its own class, you do not have to do so.

It might be handy to use in-line definition of a model. Try the following inside console:

```
$m = new \atk4\data\Model($db, 'contact_info');
$m->addFields(['address_1','address_2']);
$m->addCondition('address_1', 'not', null);
$m->loadAny();
$m->get();
$m->action('count')->getOne();
```

Next, exit and create file *src/Model_ContactInfo.php*:

```
<?php
class Model_ContactInfo extends \atk4\data\Model
{
    public $table = 'contact_info';
    function init()
    {
        parent::init();

        $this->addFields(['address_1','address_2']);
        $this->addCondition('address_1','not', null);
    }
}
```

Save, exit and run console again. You can now type this:

```
$m = new Model_ContactInfo($db);
$m->loadAny();
$m->get();
```

Note: Should the "addCondition" be located inside model definition or inside your inline code? To answer this question - think - would Model_ContactInfo have application without the condition? If yes then either use addCondition

in-line or create 2 classes.

2.3 Model State

When you create a new model object, you can change its state to perform various operations on your data. The state can be broken down into the following categories:

2.3.1 Persistence

When you create instance of a model (*new Model*) you need to specify `Persistence` as a parameter. If you don't you can still use the model, but it won't be able to `Model::load()` or `Model::save()` data.

Once model is associated with one persistence, you cannot re-associate it. Method `Model::init()` will be executed only after persistence is known, so that method may make some decisions based on chosen persistence. If you need to store model inside a different persistence, this is achieved by creating another instance of the same class and copying data over. You must however remember that any fields that you have added in-line will not be recreated.

2.3.2 DataSet (Conditions)

Model object may have one or several conditions applied. Conditions will limit which records model can load (make active) and save. Once the condition is added, it cannot be removed for safety reasons.

Suppose you have a method that converts DataSet into JSON. Ability to add conditions is your way to specify which records to operate on:

```
function myexport(\atk4\data\Model $m, $fields)
{
    return json_encode($m->export($fields));
}

$m = new Model_User($db);
$m->addCondition('country_id', '2');

myexport($m, ['id','username','country_id']);
```

If you want to temporarily add conditions, then you can either clone the model or use `Model::tryLoadBy`.

2.3.3 Active Record

Active Record is a third essential piece of information that your model stores. You can load / unload records like this:

```
$m = new Model_User($db);
$m->loadAny();

$m->get();      // inside console, this will show you what's inside your model

$m['email'] = 'test@example.com';
$m->save();
```

You can call *$m->loaded()* to see if there is active record and *$m->id* will store the ID of active record. You can also un-load the record with *$m->unload()*.

By default no records are loaded and if you modify some field and attempt to save unloaded model, it will create a new record.

Model may use some default values in order to make sure that your record will be saved inside DataSet:

```
$m = new Model_User($db);
$m->addCondition('country_id', 2);
$m['username'] = 'peter';
$m->save();

$m->get(); // will show country_id as 2
$m['country_id'] = 3;
$m->save();  // will generate exception because model you try to save doesn't match␣
↪conditions set
```

2.3.4 Other Parameters

Apart from the main 3 pieces of "state" your Model holds there can also be some other parameters such as:

- order
- limit
- only_fields

You can also define your own parameters like this:

```
$m = new Model_User($db, ['audit'=>false]);

$m->audit
```

This can be used internally for all sorts of decisions for model behavior.

Getting Started

It's time to create the first Model. Open *src/Model_User.php* which should look like this:

```
<?php
class Model_User extends \atk4\data\Model
{
    public $table = 'user';

    function init() {
        parent::init();

        $this->addField('username');
        $this->addField('email');

        $j = $this->join('contact_info', 'contact_info_id');
        $j->addField('address_1');
        $j->addField('address_2');
        $j->addField('address_3');
        $j->hasOne('country_id', 'Country');

    }
}
```

Extend either the base Model class or one of your existing classes (like Model_Client). Define $table property unless it is already defined by parent class. All the properties defined inside your model class are considered "default" you can re-define them when you create model instances:

```
$m = new Model_User($db, 'user2'); // will use a different table

$m = new Model_User($db, ['table'=>'user2']); // same
```

Note: If you're trying those lines, you will also have to create this new table inside your MySQL database:

```
create table user2 as select * from user
```

As I mentioned - `Model::init` is called when model is associated with persistence. You could create model and associate it with persistence later:

```
$m = new Model_User();

$db->add($m); // calls $m->init()
```

You cannot add conditions just yet, although you can pass in some of the defaults:

```
$m = new Model_User(['table'=>'user2']);

$db->add($m); // will use table user2
```

2.4 Adding Fields

Methods `Model::addField()` and `Model::addFields()` can declare model fields. You need to declare them before you are able to use. You might think that some SQL reverse-engineering could be good at this point, but this would mimic your business logic after your presentation logic, while the whole point of Agile Data is to separate them, so you should, at least initially, avoid using generators.

In practice, `Model::addField()` creates a new 'Field' object and then links it up to your model. This object is used to store some information about your field, but it also participates in some field-related activity.

2.5 Table Joins

Similarly, `Model::join()` creates a Join object and stores it in $j. The Join object defines a relationship between the master `Model::table` and some other table inside persistence domain. It makes sure relationship is maintained when objects are saved / loaded:

```
$j = $this->join('contact_info', 'contact_info_id');
$j->addField('address_1');
$j->addField('address_2');
```

That means that your business model will contain 'address_1' and 'address_2' fields, but when it comes to storing those values, they will be sent into a different database table and the records will be automatically linked.

Lets once again load up the console for some exercises:

```
$m = new Model_User($db);

$m->loadBy('username','john');
$m->get();
```

At this point you'll see that address has also been loaded for the user. Agile Data makes management of related records transparent. In fact you can introduce additional joins depending on class. See classes Model_Invoice and Model_Payment that join table *document* with either *payment* or *invoice*.

As you load or save models you should see actual queries in the console, that should give you some idea what kind of information is sent to the database.

Adding Fields, Joins, Expressions and References creates more objects and 'adds' them into Model (to better understand how Model can behave like a container for these objects, see documentation on Agile Core Containers). This architecture of Agile Data allows database persistence to implement different logic that will properly manipulate features of that specific database engine.

2.6 Understanding Persistence

To make things simple, console has already created persistence inside variable *$db*. Load up *console.php* in your editor to look at how persistence is set up:

```
$app->db = \atk4\data\Persistence::connect($dsn, $user, $pass);
```

The *$dsn* can also be using the PEAR-style DSN format, such as: "mysql://user:pass@db/host", in which case you do not need to specify $user and $pass.

For some persistence classes, you should use constructor directly:

```
$array = [];
$array[1] = ['name'=>'John'];
$array[2] = ['name'=>'Peter'];

$db = new \atk4\data\Persistence\Array_($array);
$m = new \atk4\data\Model($db);
$m->addField('name');
$m->load(2);
echo $m['name'];  // Peter
```

There are several Persistence classes that deal with different data sources. Lets load up our console and try out a different persistence:

```
$a=['user'=>[],'contact_info'=>[]];
$ar = new \atk4\data\Persistence\Array_($a);
$m = new Model_User($ar);
$m['username']='test';
$m['address_1']='street'

$m->save();

var_dump($a); // shows you stored data
```

This time our Model_User logic has worked pretty well with Array-only persistence logic.

Note: Persisting into Array or MongoDB are not fully functional as of 1.0 version. We plan to expand this functionality soon, see our development roadmap.

Your application normally uses multiple business entities and they can be related to each-other.

Warning: Do not mix-up business model references with database relations (foreign keys).

References are defined by calling `Model::hasOne()` or `Model::hasMany()`. You always specify destination model and you can optionally specify which fields are used for conditioning.

2.7 One to Many

Launch up console again and let's create reference between 'User' and 'System'. As per our database design - one user can have multiple 'system' records:

```
$m = new Model_User($db);
$m->hasMany('System');
```

Next you can load a specific user and traverse into System model:

```
$m->loadBy('username', 'john');
$s = $m->ref('System');
```

Unlike most ORM and ActiveRecord implementations today - instead of returning array of objects, `Model::ref()` actually returns another Model to you, however it will add one extra Condition. This type of reference traversal is called "Active Record to DataSet" or One to Many.

Your Active Record was user john and after traversal you get a model with DataSet corresponding to all Systems that belong to user john. You can use the following to see number of records in DataSet or export DataSet:

```
$s->loaded();
$s->action('count')->getOne();
$s->export();
$s->action('count')->getDebugQuery();
```

2.8 Many to Many

Agile Data also supports another type of traversal - 'DataSet to DataSet' or Many to Many:

```
$c = $m->ref('System')->ref('Client');
```

This will create a Model_Client instance with a DataSet corresponding to all the Clients that are contained in all of the Systems that belong to user john. You can examine the this model further:

```
$c->loaded();
$c->action('count')->getOne();
$c->export();
$c->action('count')->getDebugQuery();
```

By looking at the code - both MtM and OtM references are defined with 'hasMany'. The only difference is the loaded() state of the source model.

Calling ref()->ref() is also called Deep Traversal.

2.9 One to One

The third and final reference traversal type is "Active Record to Active Record":

```
$cc = $m->ref('country_id');
```

This results in an instance of Model_Country with Active Record set to the country of user john:

```
$cc->loaded();
$cc->id;
$cc->get();
```

2.10 Implementation of References

When reference is added using `Model::hasOne()` or `Model::hasMany()`, the new object is created and added into Model of class `ReferenceHasMany` or `ReferenceHasOne` (or `ReferenceHasOne_SQL` in case you use SQL database). The object itself is quite simple and you can fetch it from the model if you keep the return value of hasOne() / hasMany() or call `Model::getRef()` with the same identifier later on. You can also use `Model::hasRef()` to check if reference exists in model.

Calling `Model::ref()` will proxy into the ref() method of reference object which will in turn figure out what to do.

Additionally you can call `Model::addField()` on the reference model that will bring one or several fields from related model into your current model.

Finally this reference object contains method `Reference::getModel()` which will produce a (possibly) fresh copy of related entity and will either adjust it's DataSet or set the active record.

Since NoSQL databases will always have some specific features, Agile Data uses the concept of 'action' to map into vendor-specific operations.

2.11 Aggregation actions

SQL implements methods such as sum(), count() or max() that can offer you some basic aggregation without grouping. This type of aggregation provides some specific value from a data-set. SQL persistence implements some of the operations:

```
$m = new Model_Invoice($db);
$m->action('count')->getOne();
$m->action('fx', ['sum', 'total'])->getOne();
$m->action('fx', ['max', 'shipping'])->getOne();
```

Aggregation actions can be used in Expressions with hasMany references and they can be brought into the original model as fields:

```
$m = new Model_Client($db);
$m->getRef('Invoice')->addField('max_delivery', ['aggregate'=>'max', 'field'=>
↪'shipping']);
$m->getRef('Payment')->addField('total_paid', ['aggregate'=>'sum', 'field'=>'amount
↪']);
$m->export(['name','max_delivery','total_paid']);
```

The above code is more concise and can be used together with reference declaration, although this is how it works:

```
$m = new Model_Client($db);
$m->addExpression('max_delivery', $m->refLink('Invoice')->action('fx', ['max',
↪'shipping']));
$m->addExpression('total_paid', $m->refLink('Payment')->action('fx', ['sum', 'amount
↪']));
$m->export(['name','max_delivery','total_paid']);
```

In this example calling refLink is similar to traversing reference but instead of calculating DataSet based on Active Record or DataSet it references the actual field, making it ideal for placing into sub-query which SQL action is using. So when calling like above, action() will produce expression for calculating max/sum for the specific record of Client and those calculation are used inside an Expression().

Expression is a special type of read-only Field that uses sub-query or a more complex SQL expression instead of a physical field. (See Expressions and References)

2.12 Field-reference actions

Field referencing allows you to fetch a specific field from related model:

```
$m = new Model_Country($db);
$m->action('field', ['name'])->get();
$m->action('field', ['name'])->getDebugQuery();
```

This is useful with hasMany references:

```
$m = new Model_User($db);
$m->getRef('country_id')->addField('country', 'name');
$m->loadAny();
$m->get();  // look for 'country' field
```

hasMany::addField() again is a short-cut for creating expression, which you can also build manually:

```
$m->addExpression('country', $m->refLink('country_id')->action('field',['name']));
```

2.13 Multi-record actions

Actions also allow you to perform operations on multiple records. This can be very handy with some deep traversal to improve query efficiency. Suppose you need to change Client/Supplier status to 'suspended' for a specific user. Fire up a console once away:

```
$m = new Model_User($db);
$m->loadBy('username','john');
$m->hasMany('System');
```

(continues on next page)

(continued from previous page)

```
$c = $m->ref('System')->ref('Client');
$s = $m->ref('System')->ref('Supplier');

$c->action('update')->set('status', 'suspended')->execute();
$s->action('update')->set('status', 'suspended')->execute();
```

Note that I had to perform 2 updates here, because Agile Data considers Client and Supplier as separate models. In our implementation they happened to be in a same table, but technically that could also be implemented differently by persistence layer.

2.14 Advanced Use of Actions

Actions prove to be very useful in various situations. For instance, if you are looking to add a new user:

```
$m = new Model_User($db);
$m['username'] = 'peter';
$m['address_1'] = 'street 49';
$m['country'] = 'UK';
$m->save();
```

Normally this would not work, because country is read-only expression, however if you wish to avoid creating an intermediate select to determine ID for 'UK', you could do this:

```
$m = new Model_User($db);
$m['username'] = 'peter';
$m['address_1'] = 'street 49';
$m['country_id'] = (new Model_Country($db))->addCondition('name','UK')->action('field
↪',['id']);
$m->save();
```

This way it will not execute any code, but instead it will provide expression that will then be used to lookup ID of 'UK' when inserting data into SQL table.

Expressions that are defined based on Actions (such as aggregate or field-reference) will continue to work even without SQL (although might be more performance-expensive), however if you're stuck with SQL you can use free-form pattern-based expressions:

```
$m = new Model_Client($db);
$m->getRef('Invoice')->addField('total_purchase', ['aggregate'=>'sum', 'field'=>'total
↪']);
$m->getRef('Payment')->addField('total_paid', ['aggregate'=>'sum', 'field'=>'amount
↪']);

$m->addExpression('balance','[total_purchase]+[total_paid]');
$m->export(['name','balance']);
```

You should now be familiar with the basics of Agile Data. To find more information on specific topics, use the rest of the documentation.

Agile Data is designed in an extensive pattern - by adding more objects inside Model a new functionality can be introduced. The described functionality is never a limitation and 3rd party code or you can add features that Agile Data authors are not even considered.

CHAPTER 3

Model

3.1 Test 123

Hello world there

CHAPTER 4

Persistence Documentation

See also <model.rst>, ok?

CHAPTER 5

Fields

CHAPTER 6

References between Models

See also *Model*

6.1 Deep Traversal

CHAPTER 7

Utils

See also *Model*

7.1 DeepCopy

Deep copy

Hello `Boolean`

CHAPTER 8

Indices and tables

- genindex
- search