Agile Data

Release 1.0.1

Jun 15, 2017

Contents

Contents:

CHAPTER 1

Overview

Agile Data is a unique SQL/NoSQL access library that promotes correct Business Logic design in your PHP application and implements database access in a flexible and scalable way.

Simple to learn

We have designed Agile Data to be very friendly for those who started programming recently and teach them correct patterns through clever architectural design.

Agile Data carries the spirit of PHP language in general and gives developer ability to make choices. The framework can be beneficial even in small applications, but the true power can be drawn out of Agile Toolkit.

Fresh Concepts

In common data mapping techniques developer operates with objects that represent individual entities. Whenever he has to work with multiple records, he is presented with array of those objects.

Agile Data introduces fresh concept for "DataSet" that represent collection of entities stored inside a database. A concept of "Action" allows developer to execute operations that will affect all records in a DataSet.

Separation of Business Logic and Persistence

We educate developer about separating their Domain logic from persistence following the best practices of enterprise software. We offer the solution that works really well for most people and those who have extreme requirements can extend.

For example, you can customize persistence logic of Data Model with your own query logic where necessary.

Major Databases are Supported

The classic approach of record mapping puts low requirements on the database, but as result sacrifice performance. The abstraction of queries leads to your code being reliant on SQL vendor.

Agile Data introduces concepts that can be implemented across multiple database vendors regardless of their support for SQL. Agile Data even works with NoSQL databases like MongoDB.

If you use SQL vendor, the standard operations will be more efficient, but if you operate with a very basic database such as MemCache, then you can still simulate basic functionality.

We make it our goal to define a matrix of basic functionality and extended functionality and educate developer how to write code that is both high performance and cross-compatible.

Extensibility

Agile Data is designed to be extremely extensive. The framework already includes all the basic functionality that you would normally need, but there are more awesome things that are built as extensions:

- join support - Map your Business Model to multiple tables
- aggregate models - Build your Report Models on top of Domain models and forget about custom queries
- more vendors - Add support for more vendors and your existing application code will just work
- validation - Perform data validation on Domain level

Great for UI Frameworks

Agile Data does not approach Business Models just as property bags. Each field has some useful meta-information such as:

- type
- caption
- hint
- etc

This information is useful if you are using UI framework. Agile Toolkit takes advantage of this information to automatically populate your form / grid fields and is able to work directly with your models.

CHAPTER 2

Quickstart

Agile Data Framework is built around some unique concepts. Your knowledge of other ORM, ActiveRecord and QueryBuilder tools could be helpful, but you should carefully go through the basics if you want to know how to use Agile Data efficiently.

The distinctive goal for Agile Data is ability to "execute" complex opeartions on the database server directly, such as aggregation, sub-queries, joins and unions but only if the database server supports those operations.

Developer would normally create a declaration like this:

```
$user->hasMany('Order')->addField('total', ['aggregate'=>'sum']);
```

It is up to Agile Data to decide what's the most efficient way to implement the aggregation. Currently only SQL persistence is capable of constructing aggregate sub-query.

Requirements

If you wish to try out some examples in this guide, you will need the following:

- PHP 5.5 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
```

Console is using Psysh to help you interact with objects like this:

```
> $db
=> atk4\data\Persistence_SQL {...}

> exit
```

Note: I recommend that you enter statements into console one-by-one and carefully observe results. You should also experiment where possible, try different conditions or no conditions at all.

You can always create new model object if you mess up. If you change any of the classes, you'll have to restart console.

There seem to be a bug inside Psysh where it looses MySQL connection, in this case restart the console.

Core Concepts

Business Model (see *Working With Business Models*) 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 *Loading and Saving (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)

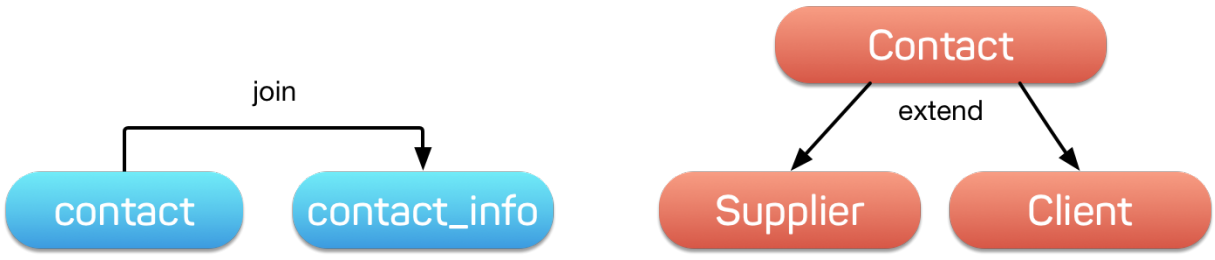
For developer, persistence should be a secondary concern, after all it is possible to switch from one persistence to another and compensate for the feature differences without major refactoring.

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

Active Record (see *Setting and Getting active record data*) 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 *Actions*) 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.

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.

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.

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:

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.

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`.

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
```

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:

```
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
```

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.

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.

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.

References between Models

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.

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();
```

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.

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();
```

Implementation of References

When reference is added using *Model::hasOne()* or `Model::hasMany()`, the new object is created and added into Model of class `Reference_Many` or *Reference_One* (or *Reference_SQL_One* 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.

Actions

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

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*)

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']));
```

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');
$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.

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

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']);
```

Conclusion

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

Introduction to Architectural Design

Layering is one of the most common techniques that software designers use to break apart a complicated software system. A modern application would have three primary layers:

- Presentation - Display of information (HTML generation, UI, API or CLI interface)
- Domain - Logic that is the real point of the system
- Data Source - Communication with databases, messaging systems, transaction managers, other packages

A persistence mechanism is a way how you save the data from some kind of in-memory model to the database. Apart from data-bases modern system also use REST services or interact with caches or files to load/store data.

Due to implementation specifics of the various data sources, making a "universal" persistence logic that can store Domain objects efficiently is not a trivial task. Various frameworks implement "Active Record", "ORM" and "Query Builder" patterns in attempts to improve data access.

The common problems when trying to simplify mapping of domain logic include:

- Performance - Traversing references where you deal with millions of related records - Executing multi-row database operation
- Reduced features - Inability to use vendor-specific features such as SQL expression syntax - Derive calculations from multi-row sub-selects - Tweak persistence-related operations
- Abstraction - Domain objects are often restricted by database schema - Difficult to use Domain objects without database connection (e.g. in Unit Tests)

Agile Data implements a fresh concepts that separates your Domain from persistence cleanly yet manages to solve problems mentioned above.

The concepts implemented by Agile Data framework may require some getting used to (especially if you used some traditional ORMs or Active Record implementations before).

Once you learn the concept behind Agile Data, you'll be able to write "Domain objects" of your application with ease through a readable code and without impact on your application performance or feature restrictions.

The Domain Layer Scope

Agile Data is a framework that will allow you to define your Domain objects and will map them into database of your choice.

You can use Agile Data with SQL (PDO-compatible) vendors, NoSQL (MongoDB) or memory Arrays. Support for other database vendors can be added through add-ons.

The Danger of Raw Queries

If you still think that writing SQL queries is the most efficient way to work with database, you are probably not considering other disadvantages of this approach:

- Parameters you specify to a query need to be escaped
- Complex queries are more difficult to write and debug
- Various parts of your application may want to change query (soft-delete add-on?)
- Optimization in your database may impact your Domain logic and even presentation
- Changing your database vendor or storing object data in cache is harder
- Difficult to maintain code

There are more problems such as difficulty in unit-testing your Domain object code.

Purity levels of Domain code

Agile Data focuses on creating "patterns" that can live in "Domain" layer. There are three levels of code "purity":

- Implement patterns for working with for Domain objects.
- Implement patterns for "persistence-backed" Domain objects.
- Implement extensions for "persisting"

Some of your code will focus on working with Domain object without any concern about "persistence". A good example is "Validation". When you Validate your Domain object you just need to check field values, you would not even care where data came from.

Most of your code, however, will assume existence of SOME "persistence", but will not rely on anything specific. Calculating total amount of your shopping basked price is such an operation. Basket items are stored somewhere - array, SQL or NoSQL and all you need is to calculate sum(amount). You don't even know how "amount" field is called in the database.

While most of relational mapping solutions would load all basket items, Agile Data performs same operations inside database if possible.

Finally - some of your code may rely of some specific database vendor features. Example would be defining an expression using "IF (expr, val1, val2) " expression for some field of Domain model or using stored procedure as the source instead of table.

Agile Data offers you ability to move as much code as possible to the level with highest "purity", but even if you have to write chunk of SQL code, you can do it without compromising cross-vendor compatibility.

Domain Logic

When dealing with Domain logic, you work with a single object.

When we start developing a new application, we first decide on the Model structure. Think what models your application will use and how they are related. Do not think in terms of "tables", but rather think in terms of "objects" and properties of those objects.

All of those model properties are "declared".

Domain Models

Congratulations, you have just designed a model layer of your application. Remember that it had nothing to do with your database structure, right?

- Client
- Order
- Admin

A code to declare a model:

```
class Model_User extends data\Model { }

class Model_Client extends Model_User { }

class Model_Admin extends Model_User { }

class Model_Order extends data\Model { }
```

Domain Model Methods

Next we need to write down various "functions" your application would have to perform and attribute those to individual models. At the same time think about object inheritance.

- User - sendPasswordReminder()
- Client (extends User) - register() - checkout()
- Admin (extends User) - showAuditLog()
- Order

Code:

```
class Model_Client extends Model_User {
    function sendPasswordReminder() {

        mail($this['email'], 'Your password is: '.$this['password']);
    }
}
```

At this stage you should not think about "saving" your entries. Think of your objects as if they would forever exist in your memory. Also don't bother with basic actions such as adding new order or deleting order.

Domain Model Fields

Our next step is to define object fields (or properties). Remember that inheritance is at play here so you can take advantage of OOP:

- User - name, is_vip, email, password, password_change_date
- Client - phone
- Admin - permission_level
- Order - description, amount, is_paid

Those fields are not just mere "properties", but have more "meta" information behind them and that's why we call them "fields" and not "properties". A typical field contain information about field name, caption, type, validation rules, persistence rules, presentation rules and more. Meta information is optional and it can be used by automated processes (such as presentation or persistence).

For instance, is_paid has a *type('boolean')* which means it will be stored as 1/0 in MySQL, but will use true/false in MongoDB. It will be displayed as a checkbox. Those decisions are made by the framework and will simplify your life, however if you want to do things differently, you will still be able to override default behavior.

Code to declare fields:

```
class Model_Order extends data\Model {
    function init() {
        parent::init();

        $this->addField('description');
        $this->addField('amount')->type('money');
        $this->addField('is_paid')->type('boolean');
    }
}
```

Code to access field values:

```
$order['amount'] = 1200.20;
```

Domain Model Relationship

Next - references. Think how those objects relate to each-other. Think in terms of "specific object" and not database relations. Client has many Orders. Order has one Client.

- User - hasMany(Client)
- Client - hasOne(User)

There are no "many-to-many" relationship in Domain Model because relationships work from a specific record, but more on that later.

Code (add inside *init()*):

```
class Model_Client extends Model_User {
    function init() {
        parent::init();

        $this->hasMany('Order');
    }
}
```

```
class Model_Order extends data\Model {
    function init() {
        parent::init();

        $this->hasOne('Client');

        // addField declarations
    }
}
```

Persistence backed Domain Logic

Once we establish that Model object and set its persistence layer, we can start accessing it. Here is the code:

```
$order = new Model_Order();
// $order is not linked with persistence


$real_order = $db->add('Model_Order');
// $real_order is associated with specific persistence layer $db
```

ID Field

Each object is stored with some unique identifier, so you can load and store object if you know it's ID:

```
$order->load(20);
$order['amount'] = 1200.20;
$order->save();
```

Persistence-specific Code

Finally, some code may rely on specific features of your persistence layer.

Domain Model Expressions

A final addition to our Domain Model are expressions. Those are the "formulas" where the value cannot be changed directly, but is actually derived from other values.

- User - is_password_expired
- Client - amount_due, total_order_amount

Here field *is_password_expired* is the type of expression that is based on the field *password_change_date* and system date. In other words the value of this expression will be different depending on parameter outside of your app.

Field *amount_due* is a sum of amount for all Orders by specific User for which condition "is_paid=false" is met. *total_order_amount* is similar, however there is no condition on the order.

With all of the above we have finished our "Domain Model" declaration. We haven't done any assumptions on where and how data is stored, which vendor we are using or how we can ensure that expressions will operate.

This is, however, a good point for you to write the initial batch of the code.

Code:

```
class Model_User extends data\Model {
    function init() {
        parent::init();

        $this->addField('password');
        $this->addField('password_change_date');


        if ($this->supports('sql-expression')) {

            $this->addExpression('is_password_expired')
                ->set(
                    '{} < NOW() - INTERVAL 1 MONTH',
                    [$this->getElement('password_change_date')]
                );
        }

    }
}
```

Persistence Hooks

Hooks can help you perform operations when object is being persisted:

```
class Model_User extends data\Model {
    function init() {
        parent::init();

        // addField() declaration
        // addExpression('is_password_expired')

        $this->addHook('beforeSave', function($m) {
            if ($m->isDirty('password')) {
                $m['password'] = encrypt_password($m['password']);
                $m['password_change_date'] = $m->expr('now()');
            }
        });
    }
}
```

DataSet Declaration

So far we have only looked at a single record - one User or one Order. In practice our application must operate with multiple records.

DataSet is an object that represents collection of Domain model records that are persisted:

```
$order = $db->add('Model_Order');
$order->load(10);
```

In scenario above we loaded a specific record. Agile Data does not create a separate object when loading, instead the same object is re-used. This is done to preserve some memory.

So in the code above *$order* is not created for the record, but it can load any record from the DataSet. Think of it as a "window" into a large table of Orders:

```
$sum = 0;
$order = $db->add('Model_Order');
$order->load(10);
$sum += $order['amount'];

$order->load(11);
$sum += $order['amount'];

$order->load(13);
$sum += $order['amount'];
```

You can iterate over the DataSet:

```
$sum = 0;
foreach($db->add('Model_Order') as $order) {
    $sum += $order['amount'];
}
```

You must remember that the code above will only create a single object and iterating it will simply make it load different values.

At this point, I'll jump ahead a bit and will show you an alternative code:

```
$sum = 0;
$sum = $db->add('Model_Order')->sum('amount')->getOne();
```

It will have same effect as the code above, but will perform operation of adding up all order amounts inside the database and save you a lot of CPU cycles.

Domain Conditions

If your database has 3 clients - 'Joe', 'Bill', and 'Steve' then the DataSet of "Client" has 3 records.

DataSet concept lives in "Domain Logic" therefore you can use it safely without worrying that you will introduce unnecessary bindings into persistence and break single-purpose principle of your objects:

```
foreach ($clients as $client) {
    // echo $client['name']."\n";
}
```

The above is a Domain Model code. It will iterate through the DataSet of "Clients" and output 3 names. You can also "narrow down" your DataSet by adding a restriction:

```
$sum = 0;
foreach($db->add('Model_Order')->addCondition('is_paid', true) as $order) {
    $sum += $order['amount'];
}
```

Related DataSets

Next, let's look on the orders of specific user. How would you load orders of a specific user. Depending on your past experience you might think about "querying" Order table with condition on user_id. We can't do that, because "query", "table" and "user_id" are persistence details and we must keep them outside of business logic. Other ORM solution give you something like this:

```
$array_of_orders = $user->orders();
```

Unfortunately this has practical performance implications and scalability constraints. What if your user is having millions of orders? Even with lazy-loading, you will be operating with million "id" records.

Agile Data implements traversal as a simple operation that converts one DataSet into another:

```
$user_dataset->addCondition('is_vip', true);
$vip_orders = $user_dataset -> refSet('Order');

$sum = $vip_orders->sum('amount')->getOne();
```

The implementation of refSet is pretty powerful - $user_dataset can address 3 users in the database and only 2 of those users are VIP. Typical ORM would require you to fetch all VIP records and then perform additional queries to find their orders.

Agile Data, however, perform traversal without accessing database at all. After *refSet()* is executed, you have a new DataSet with a condition based on user sub-query. The actual implementation may be different depending on vendor, but Agile Data will prefer not to fetch list of "user_id"s without need.

Domain Model Actions

Persistence layer in Agile Data uses intelligent mapping of your Domain Logic into DatabaseVendor-specific operations.

To continue my example from above, I'll use a query method to calculate number of orders placed by VIP clients:

```
$vip_order_count = $vip_orders->count()->getOne();
```

This code will attempt to execute a single-query only, however the ability to optimize your request relies on the capabilities of database vendor. The actual database operation(s) might look like this on SQL database:

```
select count(*) from `order` where user_id in
    (select id from user where type="user" and is_vip=1)
```

While with MongoDB, the query could be different:

```
$ids = collections.client.find({'is_vip':true}).field('id');
return collections.order.find({'user_id':$ids}).count();
```

Finally the code above will work even if you use a simple Array as a data source:

```
$db = new data\Persistence\Array([
  'client'=>[[
    'name'=>'Joe',
    'email'=>'joe@yahoo.com',
    'Orders'=>[
       ['amount'=>10], ['amount'=>20]
    ]
```

```
  ],[
    'name'=>'Bill',
    'email'=>'bill@yahoo.com',
    'Orders'=>[
       ['amount'=>35]
     ]
  ]
]);
```

So getting back to the operation above, lets look at it in more details:

```
$vip_order_count = $vip_orders->count()->getOne();
```

While "vip_orders" is actually a DataSet, executing count() will cross you over into persistence layer. However this method is returning a new object called "Action", which is then executed when you call getOne().

Even though for a brief moment you had your hands on a "database-vendor specific" object, you have immediately converted Action into an actual value. As result your code is universal and is not persistence-specific. In Agile Data we permit code like that in our Domain Model and we call it "Domain Model Action".

Let me define this properly: Domain Model Action is an operation that can be executed in your Domain Model layer which assumes existence of SOME Persistence for your model, but not a specific one.

As long as your Domain Model is restricted to generic Domain Model Actions, it will not violate SRP (Single Responsibility Principle)

Unique Features of Persistence Layer

More often than not, your application is designed and built with a specific persistence layer in mind. If you are using SQL database, you want to

Before we talk "databases", we must outline a few challenges:

- our business model described above should work with various database vendors
- we should be able to perform basic Unit tests on our domain logic
- single vs multiple records
- ..add more..

CHAPTER 4

Working With Business Models

Note: This documentation needs to be reworked to be easier to read!

class Model

Initialization

Model class implements your Business Model - single entity of your business logic. When you plan your business application you should create classes for all your possible business entities by extending from "Model" class.

`Model::init()`

Method init() will automatically be called when your Model is associated with persistence driver. Use it to define fields of your model:

```
class Model_User extends atk4\data\Model
{
    function init() {
        parent::init();

        $this->addField('name');
        $this->addField('surname');
    }
}
```

`Model::addField`(*$name*, *$defaults*)

Creates a new field objects inside your model (by default the class is 'Field'). The fields are implemented on top of Containers from Agile Core.

Second argument to addField() can supply field default properties:

```
$this->addField('surname', ['default'=>'Smith']);
```

Read more about *Field*

Populating Data

Model::**insert**(*$row*)
: Inserts a new record into the database and returns $id. It does not affect currently loaded record and in practice would be similar to:

```
$m_x = $m;
$m_x->unload();
$m_x->set($row);
$m_x->save();
return $m_x;
```

The main goal for insert() method is to be as fast as possible, while still performing data validation. After inserting method will return cloned model.

Model::**import**(*$data*)
: Similar to insert() however works across array of rows. This method will not return any IDs or models and is optimized for importing large amounts of data.

The method will still convert the data needed and operate with joined tables as needed. If you wish to access tables directly, you'll have to look into Persistence::insert($m, $data, $table);

Associating Model with Database

Normally you should always associate your model with persistence layer (database) when you create the instance like this:

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

property Model::$**persistence**
: Refers to the persistence driver in use by current model. Calling certain methods such as save(), addCondition() or action() will rely on this property.

property Model::$**persistence_data**
: Array containing arbitrary data by a specific persistence layer.

property Model::$**table**
: If $table property is set, then your persistence driver will use it as default table / collection when loading data. If you omit the table, you should specify it when associating model with database:

```
$m = new Model_User($db, 'user');
```

Model::**withPersistence**(*$persistence*, *$id = null*, *$class = null*)
: Creates a duplicate of a current model and associate new copy with a specified persistence. This method is useful for moving model data from one persistence to another.

Working with selective fields

When you normally work with your model then all fields are available and will be loaded / saved. You may, however, specify that you wish to load only a sub-set of fields.

(In ATK4.3 we call those fields "Actual Fields")

`Model::onlyFields(`*$fields*`)`
Specify array of fields. Only those fields will be accessible and will be loaded / saved. Attempt to access any other field will result in exception.

`Model::allFields()`
Restore to full set of fields. This will also unload active record.

property `Model::$only_fields`
Contains list of fields to be loaded / accessed.

Setting and Getting active record data

When your record is loaded from database, record data is stored inside the $data property:

property `Model::$data`
Contains the data for an active record.

Model allows you to work with the data of single a record directly. You should use the following syntax when accessing fields of an active record:

```
$m['name'] = 'John';
$m['surname'] = 'Peter';
```

When you modify active record, it keeps the original value in the $dirty array:

`Model::set()`
Set field to a specified value. The original value will be stored in $dirty property. If you pass non-array, then the value will be assigned to the *Title Field*.

`Model::unset()`
Restore field value to it's original:

```
$m['name'] = 'John';
echo $m['name']; // John

unset($m['name']);
echo $m['name']; // Original value is shown
```

This will restore original value of the field.

`Model::get()`
Returns one of the following:

- If value was set() to the field, this value is returned
- If field was loaded from database, return original value
- if field had default set, returns default
- returns null.

`Model::isset()`
Return true if field contains unsaved changes (dirty):

```
isset($m['name']); // returns false
$m['name'] = 'Other Name';
isset($m['name']); // returns true
```

`Model::isDirty()`
Return true if one or multiple fields contain unsaved changes (dirty):

```
if ($m->isDirty(['name','surname'])) {
    $m['full_name'] = $m['name'].' '.$m['surname'];
}
```

When the code above is placed in beforeSave hook, it will only be executed when certain fields have been changed. If your recalculations are expensive, it's pretty handy to rely on "dirty" fields to avoid some complex logic.

property `Model::$dirty`
Contains list of modified fields since last loading and their original values.

Full example:

```
$m = new Model_User($db, 'user');

// Fields can be added after model is created
$m->addField('salary', ['default'=>1000]);

echo isset($m['salary']);   // false
echo $m['salary'];          // 1000

// Next we load record from $db
$m->load(1);

echo $m['salary'];          // 2000 (from db)
echo isset($m['salary']);   // false, was not changed

$m['salary'] = 3000;

echo $m['salary'];          // 3000 (changed)
echo isset($m['salary']);   // true

unset($m['salary']);        // return to original value

echo $m['salary'];          // 2000
echo isset($m['salary']);   // false

$m['salary'] = 3000;
$m->save();

echo $m['salary'];          // 3000 (now in db)
echo isset($m['salary']);   // false
```

protected normalizeFieldName
Verify and convert first argument got get / set;

Title Field and ID Field

Those are to properties that you can specify in the model or pass it through defaults:

```
class MyModel ..
    public $title_field = 'full_name';
```

or as defaults:

```
$m = new MyModel($db, ['title_field'=>'full_name']);
```

ID Field

property Model::$**id_field**
: If your data storage uses field different than `id` to keep the ID of your records, then you can specify that in $id_field property.

Tip: You can change ID field of the current ID field by calling:

```
$m['id'] = $new_id;
$m->save();
```

This will update existing record with new $id. If you want to save your current field over another existing record then:

```
$m->id = $new_id;
$m->save();
```

You must remember that only dirty fields are saved, though. (We might add replace() function though).

Title Field

property Model::$**title_field**
: This field by default is set to 'name' will act as a primary title field of your table. This is especially handy if you use model inside UI framework, which can automatically display value of your title field in the header, or inside drop-down.

  If you don't have field 'name' but you want some other field to be title, you can specify that in the property. If title_field is not needed, set it to false or point towards a non-existent field.

  See: :php:meth::*hasOne::addTitle()* and :php:meth::*hasOne::withTitle()*

Hooks

- beforeSave [not currently working]
  - beforeInsert [only if insert] - beforeInsertQuery [sql only] (query) - afterInsertQuery (query, statement)
  - beforeUpdate [only if update] - beforeUpdateQuery [sql only] (query) - afterUpdateQuery (query, statement)
  - afterUpdate [only if existing record]
  - afterInsert [only if new record]
  - beforeUnload
  - afterUnload
- afterSave

How to verify Updates

The model is only being saved if any fields have been changed (dirty). Sometimes it's possible that the record in the database is no longer available and your update() may not actually update anything. This does not normally generate an error, however if you want to actually make sure that update() was effective, you can implement this through a hook:

```
$m->addHook('afterUpdateQuery',function($m, $update, $st) {
    if (!$st->rowCount()) {
        throw new \atk4\core\Exception([
            'Update didn\'t affect any records',
            'query'      => $update->getDebugQuery(false),
            'statement'  => $st,
            'model'      => $m,
            'conditions' => $m->conditions,
        ]);
    }
});
```

How to prevent actions

In some cases you want to prevent default actions from executing. Suppose you want to check 'memcache' before actually loading the record from the database. Here is how you can implement this functionality:

```
$m->addHook('beforeLoad',function($m, $id) {
    $data = $m->app->cacheFetch($m->table, $id);
    if ($data) {
        $m->data = $data;
        $m->id = $id;
        $m->breakHook(false);
    }
});
```

$app property is injected through your $db object and is passed around to all the models. This hook, if successful, will prevent further execution of other beforeLoad hooks and by specifying argument as 'false' it will also prevent call to $persistence for actual loading of the data.

Similarly you can prevent deletion if you wish to implement soft-delete or stop insert/modify from occurring.

CHAPTER 5

Typecasting

Typecasting is envoked when you are attempting to save or load the record. Unlike strict types and normalization, typecasting is a persistence-specific operation. Here is the sequence and sample:

```
$m->addField('birthday', ['type'=>'date']);
// type has a number of pre-defined values. Using 'date'
// instructs AD that we will be using it for staring dates
// through 'DateTime' class.

$m['birthday'] = 'Jan 1 1960';
// If non-compatible value is provided, it will be converted
// into a proper date through Normalization process. After
// this line value of 'birthday' field will be DateTime.

$m->save();
// At this point typecasting converts the "DateTime" value
// into UTC date-time representation for SQL or "MongoDate"
// type if you're persisting with MongoDB. This does not affect
// value of a model field.
```

Typecasting is necessary to save the values inside the database and restore them back just as they were before. When modifying a record, typecasting will only be invoked on the fields which were dirty.

The purpose of a flexible typecasting system is to allow you to store your date in a compatible format or even fine-tune it to match your database settings (e.g. timezone) without affecting your domain code.

You must remember that type-casting is a two-way operation. If you are introducing your own types, you will need to make sure they can be saved and loaded correctly.

Some formats such as *date*, *time* and *datetime* may have additional options to it:

```
$m->addField('registered', [
    'type'=>'date',
    'persist_format'=>'d/m/Y',
    'persist_timezone'=>'IST'
]);
```

Here is another example with booleans:

```
$m->addField('is_married', [
    'type' => 'boolean',
    'enum' => ['No', 'Yes']
]);

$m['is_married'] = 'Yes';  // normalizes into true
$m['is_married'] = true;   // better way because no need to normalize

$m->save();   // stores as "Yes" because of type-casting
```

Value types

Any type can have a value of *null*:

```
$m['is_married'] = null;
if (!$m['is_married']) {
    // either null or false
}
```

If value is passed which is not compatible with field type, Agile Data will try to normalize value:

```
$m->addField('age', ['type'=>'integer']);
$m->addField('name', ['type'=>'string']);

$m['age'] = '49.80';
$m['name'] = '       John';

echo $m['age']; // 49 - normalization cast value to integer
echo $m['name']; // 'John' - normalization trims value
```

Undefined type

If you do not set type for a field, Agile Data will not normalize and type-cast its value.

Because of the loose PHP types, you can encounter situations where undefined type is changed from *'4'* to *4*. This change is still considered "dirty".

If you use numeric value with a type-less field, the response from SQL does not distinguish between integers and strings, so your value will be stored as "string" inside the model.

The same can be said about forms, which submit all their data through POST request that has no types, so undefined type fields should work relatively good with the standard setup of Agile Data + Agile Toolkit + SQL.

Type of IDs

Many databases will allow you to use different types for ID fields. In SQL the 'id' column will usually be "integer", but sometimes it can be of a different type.

The same applies for references ($m->hasOne()).

Supported types

- 'string' - for storing short strings, such as name of a person. Normalize will trim the value.
- 'boolean' - normalize will cast value to boolean.
- 'integer' - normalize will cast value to integer.
- 'money' - normalize will round value with 4 digits after dot.
- 'float' - normalize will cast value to float.
- 'date' - normalize will convert value to DateTime object.
- 'datetime' - normalize will convert value to DateTime object.
- 'time' - normalize will convert value to DateTime object.
- 'array' - no normalization by default
- 'object' - no normalization by default

Types and UI

UI framework such as Agile Toolkit will typically rely on field type information to properly present data for views (forms and tables) without you having to explicitly specify the *ui* property.

Serialization

Some types cannot be stored natively. For example, generic objects and arrays have no native type in SQL database. This is where serialization feature is used.

Field may use serialization to further encode field value for the storage purpose:

```
$this->addField('private_key', [
    'serialize'=>'base64',
    'system'=>true,
]);
```

This is one way to store binary data. Type is unspecified but the binary value of a field will be encoded with base64 before storing and automatically decoded when you load this value back from persistence.

Supported algorithms

- 'serialize' - for storing PHP objects, uses *serialize*, *unserialize*
- 'json' - for storing objects and arrays, uses *json_encode*, *json_decode*
- 'base64' - for storing encoded strings, uses *base64_encode*, *base64_decode*
- [serialize_callback, unserialize_callback] - for custom serialization

Storing unsupported types

Here is another example defining the field that stores monetary value containing both the amount and the currency. The domain model will use an object and we are specifying our callbacks for converting:

```
$money_encode = function($x) {
    return $x->amount.' '.$x->currency;
}

$money_dencode = function($x) {
    list($amount, $currency) = explode(' ', $x);
    return new MyMoney($amount, $currency);
}

$this->addField('money', [
    'serialize' => [$money_encode, $money_decode],
]);
```

Array and Object types

Some types may require serialisaion for some persistences, for instance types 'array' and 'object' cannot be stored in SQL natively. That's why they will use *json_encode* and *json_decode* by default. If you specify a different serialisation technique, then it will be used instead of JSON.

This is handy when mapping JSON data into native PHP structures.

CHAPTER 6

Loading and Saving (Persistence)

class Model

In order to load and store data of your model inside the database your model should be "associated" with persistence layer.

Associating with Persistence

Create your persistence object first:

```
$db = \atk4\data\Persistence::connect($dsn);
```

There are several ways to link your model up with the persistence:

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

$m = $db->add(new Model_Invoice());

$m = $db->add('Invoice');
```

`Model::load()`
: Load active record from the DataSet:

```
$m->load(10);
echo $m['name'];
```

If record not found, will throw exception.

`Model::save(`*$data* `=[])`
: Store active record back into DataSet. If record wasn't loaded, store it as a new record:

```
$m->load(10);
$m['name'] = 'John';
$$m->save();
```

You can pass argument to save() to set() and save():

```
$m->unload();
$m->save(['name'=>'John']);
```

Save, like set() support title field:

```
$m->unload();
$m->save('John');
```

`Model::tryLoad()`
Same as load() but will silently fail if record is not found:

```
$m->tryLoad(10);
$m->set($data);

$m->save();     // will either create new record or update existing
```

`Model::loadAny()`
Attempt to load any matching record. You can use this in conjunction with setOrder():

```
$m->loadAny();
echo $m['name'];
```

`Model::tryLoadAny()`
Attempt to load any record, but silently fail if there are no records in the DataSet.

`Model::unload()`
Remove active record and restore model to default state:

```
$m->load(10);
$m->unload();

$m['name'] = 'New User';
$m->save();          // creates new user
```

`Model::delete(`*$id = null*`)`
Remove current record from DataSet. You can optionally pass ID if you wish to delete a different record. If you pass ID of a currently loaded record, it will be unloaded.

Inserting Record with a specific ID

When you add a new record with save(), insert() or import, you can specify ID explicitly:

```
$m['id'] = 123;
$m->save();

// or $m->insert(['Record with ID=123', 'id'=>123]);
```

However if you change the ID for record that was loaded, then your database record will also have its ID changed. Here is example:

```
$m->load(123);
$m[$m->id_field] = 321;
$m->save();
```

After this your database won't have a record with ID 123 anymore.

Type Converting

PHP operates with a handful of scalar types such as integer, string, booleans etc. There are more advanced types such as DateTime. Finally user may introduce more useful types.

Agile Data ensures that regardless of the selected database, types are converted correctly for saving and restored as they were when loading:

```
$m->addField('is_admin', ['type'=>'boolean']);
$m['is_admin'] = false;
$m->save();

// SQL database will actually store `0`

$m->load();

$m['is_admin'];  // converted back to `false`
```

Behind a two simple lines might be a long path for the value. The various components are essential and as developer you must understand the full sequence:

```
$m['is_admin'] = false;
$m->save();
```

Strict Types an Normalization

PHP does not have strict types for variables, however if you specify type for your model fields, the type will be enforced.

Calling "set()" or using array-access to set the value will start by casting the value to an appropriate data-type. If it is impossible to cast the value, then exception will be generated:

```
$m['is_admin'] = "1"; // OK, but stores as `true`

$m['is_admin'] = 123; // throws exception.
```

It's not only the 'type' property, but 'enum' can also imply restrictions:

```
$m->addField('access_type', ['enum' => ['read_only', 'full']]);

$m['access_type'] = 'full'; // OK
$m['access_type'] = 'half-full'; // Exception
```

There are also non-trivial types in Agile Data:

```
$m->addField('salary', ['type' => 'money']);
$m['salary'] = "20";  // converts to 20.00

$m->addField('date', ['type' => 'date']);
$m['date'] = time();  // converts to DateTime class
```

Finally, you may create your own custom field types that follow a more complex logic:

```
$m->add(new Field_Currency(), 'balance');
$m['balance'] = '12,200.00 EUR';
```

```
// May transparently work with 2 columns: 'balance_amount' and
// 'balance_currency_id' for example.
```

The process of converting field values as indicated above is called "normalization" and it is controlled by two model properties:

```
$m->strict_types = true;
$m->load_normalization = false;
```

Setting `Model::strict_types` to false, will still disable any type-casting and store exact values you specify regardless of type. If you switch on `Model::load_normalization` then the values will also be normalized as they are loaded from the database. Normally you should only do that if you're storing values into database by other means and not through Agile Data.

Final field flag that is worth mentioning is called `Field::read_only` and if set, then value of a field may not be modified directly:

```
$m->addField('ref_no', ['read_only' => true]);
$m->load(123);

$m['ref_no']; // perfect for reading field that is populated by trigger.

$m['ref_no'] = 'foo'; // exception
```

Note that *read_only* can still have a default value:

```
$m->addField('created', [
    'read_only' => true,
    'type'      => 'datetime',
    'default'   => new DateTime()
]);

$m->save();  // stores creation time just fine and also will loade it.
```

Note: If you have been following our "Domain" vs "Persistence" then you can probably see that all of the above functionality described in this section apply only to the "Domain" model.

Typecasting

For full documentation on type-casting see typecasting

Validation

Validation in application always depends on business logic For example, if you want *age* field to be above *14* for the user registration you may have to ask yourself some questions:

- Can user store *12* inside a age field?
- If yes, Can user persist age with value of *12*?
- If yes, Can user complete registration with age of *12*?

If 12 cannot be stored at all, then exception would be generated during set(), before you even get a chance to look at other fields.

If storing of *12* in the model field is OK validation can be called from beforeSave() hook. This might be a better way if your validation rules depends on multiple field conditions which you need to be able to access.

Finally you may allow persistence to store *12* value, but validate before a user-defined operation. *completeRegistration* method could perform the validation. In this case you can create a confirmation page, that actually stores your incomplete registration inside the database.

You may also make a decision to store registration-in-progress inside a session, so your validation should be aware of this logic.

Agile Data relies on 3rd party validation libraries, and you should be able to find more information on how to integrate them.

Multi-column fields

Lets talk more about this currency field:

```
$m->add(new Field_Currency(), 'balance');
$m['balance'] = '12,200.00 EUR';
```

It may be designed to split up the value by using two fields in the database: *balance_amount* and *balance_currency_id*. Both values must be loaded otherwise it will be impossible to re-construct the value.

On other hand, we would prefer to hide those two columns for the rest of application.

Finally, even though we are storing "id" for the currency we want to make use of References.

Your init() method for a Field_Currency might look like this:

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

    $this->never_persist = true;

    $f = $this->short_name; // balance

    $this->owner->addField(
        $f.'_amount',
        ['type' => 'money', 'system' => true]
    );

    $this->owner->hasOne(
        $f.'_currency_id',
        [
            $this->currency_model ?: new Currency(),
            'system' => true,
        ]
    );
}
```

There are more work to be done until Field_Currency could be a valid field, but I wanted to draw your attention to the use of field flags:

- system flag is used to hide *balance_amount* and *balance_currency_id* in UI.
- never_persist flag is used because there are no *balance* column in persistence.

Type Matrix

type	aliases(es)	description	native	sql	mongo
string		Will be trim() ed.			
int	integer	will cast to int make sure it's not passed as a string.	-394 , "49"	49	49
float		decimal number with floating point	3.2884,		
money		Will convert loosly-specified currency into float or dedicated format for storage. Optionally support 'fmt' proprety.	"£3, 294.48", 3.999	38294. 48 , 4	
bool	boolean	true / false type value. Optionally specify 'enum'=>['N','Y'] to store true as 'Y' and false as 'N'. By default uses [0,1].	true	1	true
array		Optionally pass 'fmt' option, which is 'json' by default. Will json_encode and json_decode(..., true) the value if database does not support array storage.	[2=> "bar "]	{2 :" bar" }	stored as- is
binary		Supports storage of binary data like BLOBs			

- Money: http://php.net/manual/en/numberformatter.parsecurrency.php.
- money: See also http://www.thefinancials.com/Default.aspx?SubSectionID=curformat

Dates and Time

There are 4 date formats supported:

- ts (or timestamp): Stores in database using UTC. Defaults into unix timestamp (int) in PHP.
- date: Converts into YYYY-MM-DD using UTC timezone for SQL. Defaults to DateTime() class in PHP, but supports string input (parsed as date in a current timezone) or unix timestamp.
- time: converts into HH:MM:SS using UTC timezone for storing in SQL. Defaults to DateTime() class in PHP, but supports string input (parsed as date in current timezone) or unix timestamp. Will discard date from timestamp.
- datetime: stores both date and time. Uses UTC in DB. Defaults to DateTime() class in PHP. Supports string input parsed by strtotime() or unix timestamp.

Customizations

Process which converts field values in native PHP format to/from database-specific formats is called typecasting. Persistence driver implements a necessary type-casting through the following two methods:

`typecastLoadRow($model, $row);`
Convert persistence-specific row of data to PHP-friendly row of data.

`typecastSaveRow($model, $row);`
Convert native PHP-native row of data into persistence-specific.

Row persisting may rely on additional methods, such as:

typecastLoadField(Field $field, $value);
Convert persistence-specific row of data to PHP-friendly row of data.

typecastSaveField(Field $field, $value);
Convert native PHP-native row of data into persistence-specific.

Duplicating and Replacing Records

In normal operation, once you store a record inside your database, your interaction will always update this existing record. Sometimes you want to perform operations that may affect other records.

Create copy of existing record

Model::**duplicate**(*$id = null*)
Normally, active record stores "id", but when you call duplicate() it forgets current ID and as result it will be inserted as new record when you execute *save()* next time.

If you pass the *$id* parameter, then the new record will be saved under a new ID:

```
// First, lets delete all records except 123
(clone $m)->addCondition('id', '!=', 123)->action('delete')->execute();

// Next we can duplicate
$m->load(123)->duplicate()->save();

// Now you have 2 records:
// one with ID=123 and another with ID={next db generated id}
echo $m->action('count')->getOne();
```

Duplicate then save under a new ID

Assuming you have 2 different records in your database: 123 and 124, how can you take values of 123 and write it on top of 124?

Here is how:

```
$m->load(123)->duplicate(124)->replace();
```

Now the record 124 will be replaced with the data taken from record 123. For SQL that means calling 'replace into x'.

Warning: You might be wondering how join() logic would work. Well there are no special treatment for joins() when duplicating records, so your new record will end up referencing a same joined record. If join is reverse, then your new record may not load.

This will be properly addressed in future versions of Agile Data.

Working with Multiple DataSets

When you load a model, conditions are applied that make it impossible for you to load record from outside of a data-set. In some cases you do want to store the model outside of a data-set. This section focuses on various use-cases like that.

Cloning versus New Instance

When you clone a model, the new copy will inherit pretty much all the conditions and any in-line modifications that you have applied on the original model. If you decide to create new instance, it will provide a *vanilla* copy of model without any in-line modifications. This can be used in conjunction to escape data-set.

`Model::newInstance` (*$class* = *null*, *$options* = [ ])

Looking for duplicates

We have a model 'Order' with a field 'ref', which must be unique within the context of a client. However, orders are also stored in a 'Basket'. Consider the following code:

```
$basket->ref('Order')->insert(['ref'=>123]);
```

You need to verify that the specific client wouldn't have another order with this ref, how do you do it?

Start by creating a beforeSave handler for Order:

```
$this->addHook('beforeSave', function($m) {
    if ($this->isDirty('ref')) {

        if (
            $m->newInstance()
                ->addCondition('client_id', $m['client_id'])
                ->tryLoadBy('ref', $m['ref'])
                ->loaded()
        ) {
            throw new Exception([
                'Order with ref already exists for this client',
                'client' => $this['client_id'],
                'ref'    => $this['ref']
            ]);
        }
    }
});
```

Important: Always use $m, don't use $this, or cloning models will glitch.

So to review, we used newInstance() to create new copy of a current model. It is important to note that newInstance() is using get_class($this) to determine the class.

Archiving Records

In this use case you are having a model 'Order', but you have introduced the option to archive your orders. The method *archive()* is supposed to mark order as archived and return that order back. Here is the usage pattern:

```
$o->addCondition('is_archived', false); // to restrict loading of archived orders
$o->load(123);
$archive = $o->archive();
$archive['note'] .= "\nArchived on $date.";
$archive->save();
```

With Agile Data API building it's quite common to create a method that does not actually persist the model.

The problem occurs if you have added some conditions on the $o model. It's quite common to use $o inside a UI element and exclude Archived records. Because of that, saving record as archived may cause exception as it is now outside of the result-set.

There are two approaches to deal with this problem. The first involves disabling after-save reloading:

```
function archive() {
    $this->reload_after_save = false;
    $this['is_archived'] = true;
    return $this;
}
```

After-save reloading would fail due to *is_archived = false* condition so disabling reload is a hack to get your record into the database safely.

The other, more appropriate option is to re-use a vanilla Order record:

```
function archive() {
    $this->save(); // just to be sure, no dirty stuff is left over

    $archive = $this->newInstance();
    $archive->load($this->id);
    $archive['is_archived'] = true;

    $this->unload(); // active record is no longer accessible

    return $archive;
}
```

This method may still not work if you extend and use "ActiveOrder" as your model. In this case you should pass the class to newInstance():

```
$archive = $this->newInstance('Order');
// or
$archive = $this->newInstance(new Order());
// or with passing some default properties:
$archive = $this->newInstance([new Order(), 'audit'=>true]);
```

In this case newInstance() would just associate passed class with the persistence pretty much identical to:

```
$archive = new Order($this->persistence);
```

The use of newInstance() however requires you to load the model which is an extra database query.

Using Model casting and saveAs

There is another method that can help with escaping the DataSet that does not involve record loading:

`Model::asModel` (*$class = null*, *$options =*[])
: Changes the class of a model, while keeping all the loaded and dirty values.

The above example would then work like this:

```
function archive() {
    $this->save(); // just to be sure, no dirty stuff is left over

    $archive = $o->asModel('Order');
    $archive['is_archived'] = true;

    $this->unload(); // active record is no longer accessible.

    return $archive;
}
```

Note that after saving 'Order' it may attempt to load_after_save just to ensure that stored model is a valid 'Order'.

`Model::saveAs` (*$class* = *null*, *$options*=[])
Save record into the database, using a different class for a model.

As in my archiving example, here is how we can eliminate need of archive() method altogether:

```
$o = new ActiveOrder($db);
$o->load(123);

$o->set(['is_arhived', true])->saveAs('Order');
```

Currently the implementation of saveAs is rather trivial, but in the future versions of Agile Data you may be able to do this:

```
// MAY NOT WORK YET
$o = new ActiveOrder($db);
$o->load(123);

$o->saveAs('ArchivedOrder');
```

Of course - instead of using 'Order' you can also specify the object with *new Order()*.

Working with Multiple Persistences

Normally when you load the model and save it later, it ends up in the same database from which you have loaded it. There are cases, however, when you want to store the record inside a different database. As we are looking into use-cases, you should keep in mind that with Agile Data Persistence can be pretty much anything including 'RestAPI', 'File', 'Memcache' or 'MongoDB'.

Important: Instance of a model can be associated with a single persistence only. Once it is associated, it stays like that. To store a model data into a different persistence, a new instance of your model will be created and then associated with a new persistence.

`Model::withPersistence` (*$persistence*, *$id* = *null*, *$class* = *null*)

Creating Cache with Memcache

Assuming that loading of a specific items from the database is expensive, you can opt to store them in a MemCache. Caching is not part of core functionality of Agile Data, so you will have to create logic yourself, which is actually quite simple.

You can use several designs. I will create a method inside my application class to load records from two persistences that are stored inside properties of my application:

```
function loadQuick($class, $id) {

    // first, try to load it from MemCache
    $m = $this->mdb->add(clone $class)->tryLoad($id);

    if (!$m->loaded()) {

        // fall-back to load from SQL
        $m = $this->sql->add(clone $class)->load($id);

        // store into MemCache too
        $m = $m->withPersistence($this->mdb)->replace();
    }

    $m->addHook('beforeSave', function($m){
        $m->withPersistence($this->sql)->save();
    });

    $m->addHook('beforeDelete', function($m){
        $m->withPersistence($this->sql)->delete();
    });

    return $m;
}
```

The above logic provides a simple caching framework for all of your models. To use it with any model:

```
$m = $app->loadQuick(new Order(), 123);

$m['completed'] = true;
$m->save();
```

To look in more details into the actual method, I have broken it down into chunks:

```
// first, try to load it from MemCache:
$m = $this->mdb->add(clone $class)->tryLoad($id);
```

The $class will be an uninitialized instance of a model (although you can also use a string). It will first be associated with the MemCache DB persistence and we will attempt to load a corresponding ID. Next, if no record is found in the cache:

```
if (!$m->loaded()) {

    // fall-back to load from SQL
    $m = $this->sql->add(clone $class)->load($id);

    // store into MemCache too
    $m = $m->withPersistence($this->mdb)->replace();
}
```

Load the record from the SQL database and store it into $m. Next, save $m into the MemCache persistence by replacing (or creating new) record. The *$m* at the end will be associated with the MemCache persistence for consistency with cached records. The last two hooks are in order to replicate any changes into the SQL database also:

```
$m->addHook('beforeSave', function($m){
    $m->withPersistence($this->sql)->save();
});

$m->addHook('beforeDelete', function($m){
    $m->withPersistence($this->sql)->delete();
});
```

I have too note that withPersistence() transfers the dirty flags into a new model, so SQL record will be updated with the record that you have modified only.

If saving into SQL is successful the memcache persistence will be also updated.

Using Read / Write Replicas

In some cases your application have to deal with read and write replicas of the same database. In this case all the operations would be done on the read replica, except for certain changes.

In theory you can use hooks (that have option to cancel default action) to create a comprehensive system-wide solution, I'll illustrate how this can be done with a single record:

```
$m = new Order($read_replica);

$m['completed'] = true;

$m->withPersistence($write_replica)->save();
$m->dirty = [];

// Possibly the update is delayed
// $m->reload();
```

By changing 'completed' field value, it creates a dirty field inside *$m*, which will be saved inside a *$write_replica*. Although the proper approach would be to reload the *$m*, if there is chance that your update to a write replica may not propagate to read replica, you can simply reset the dirty flags.

If you need further optimization, make sure *reload_after_save* is disabled for the write replica:

```
$m->withPersistence($write_replica, null, ['reload_after_save'=>false])->save();
```

or use:

```
$m->withPersistence($write_replica)->saveAndUnload();
```

Archive Copies into different persistence

If you wish that every time you save your model the copy is also stored inside some other database (for archive purposes) you can implement it like this:

```
$m->addHook('beforeSave', function($m) {
    $arc = $this->withPersistence($m->app->archive_db, false);

    // add some audit fields
    $arc->addField('original_id')->set($this->id);
    $arc->addField('saved_by')->set($this->app->user);
```

```
    $arc->saveAndUnload();
});
```

When passing 2nd argument of *false* to the withPersistence() method, it will not re-use current ID instead creating new records every time.

Store a specific record

If you are using authentication mechanism to log a user in and you wish to store his details into Session, so that you don't have to reload every time, you can implement it like this:

```
if (!isset($_SESSION['ad'])) {
    $_SESSION['ad'] = []; // initialize
}

$sess = new \atk4\data\Persistence_Array($_SESSION['ad']);
$logged_user = new User($sess);
$logged_user->load('active_user');
```

This would load the user data from Array located inside a local session. There is no point storing multiple users, so I'm using id='active_user' for the only user record that I'm going to store there.

How to add record inside session, e.g. log the user in? Here is the code:

```
$u = new User($db);
$u->load(123);

$u->withPersistence($sess, 'active_user')->save();
```

Actions

Action is a multi-row operation that will affect all the records inside DataSet. Actions will not affect records outside of DataSet (records that do not match conditions)

`Model::action`(*$action*, *$args* = [])

Prepares a special object representing "action" of a persistence layer based around your current model:

```
$m = Model_User();
$m->addCondition('last_login', '<', date('Y-m-d', strtotime('-2 months')));

$m->action('delete')->execute();
```

Action Types

Actions can be grouped by their result. Some action will be executed and will not produce any results. Others will respond with either one value or multiple rows of data.

- no results
- single value
- single row
- single column

- array of hashes

Action can be executed at any time and that will return an expected result:

```
$m = Model_Invoice();
$val = $m->action('count')->getOne();
```

Most actions are sufficiently smart to understand what type of result you are expecting, so you can have the following code:

```
$m = Model_Invoice();
$val = $m->action('count')();
```

When used inside the same Persistence, sometimes actions can be used without executing:

```
$m = Model_Product($db);
$m->addCondition('name', $product_name);
$id_query_action = $m->action('getOne',['id']);

$m = Model_Invoice($db);
$m->insert(['qty'=>20, 'product_id'=>$id_query_action]);
```

Insert operation will check if you are using same persistence. If the persistence object is different, it will execute action and will use result instead.

Being able to embed actions inside next query allows Agile Data to reduce number of queries issued.

The default action type can be set when executing action, for example:

```
$a = $m->action('field', 'user', 'getOne');

echo $a();    // same as $a->getOne();
```

SQL Actions

The following actions are currently supported by Persistence_SQL:

- select - produces query that returns DataSet (array of hashes)
- delete - produces query for deleting DataSet (no result)

The following two queries returns un-populated query, which means if you wish to use it, you'll have to populate it yourself with some values:

- insert - produces an un-populated insert query (no result).
- update - produces query for updating DataSet (no result)

Example of using update:

```
$m = Model_Invoice($db);
$m->addCondition('has_discount', true);

$m->action('update')
    ->set('has_dicount', false)
    ->execute();
```

You must be aware that set() operates on a DSQL object and will no longer work with your model fields. You should use the object like this if you can:

```
$m->action('update')
    ->set($m->getElement('has_discount'), false)
    ->execute();
```

See $actual for more details.

There are ability to execute aggregation functions:

```
echo $m->action('fx', ['max', 'salary'])->getOne();
```

and finally you can also use count:

```
echo $m->action('count')->getOne();
```

SQL Actions on Linked Records

In conjunction with Model::refLink() you can produce expressions for creating sub-selects. The functionality is nicely wrapped inside Field_SQL_Many::addField():

```
$client->hasMany('Invoice')
    ->addField('total_gross', ['aggregate'=>'sum', 'field'=>'gross']);
```

This operation is actually consisting of 3 following operations:

1. Related model is created and linked up using refLink that essentially places a condition between $client and $invoice assuming they will appear inside same query.
2. Action is created from $invoice using 'fx' and requested method / field.
3. Expression is created with name 'total_gross' that uses Action.

Here is a way how to intervene with the process:

```
$client->hasMany('Invoice');
$client->addExpression('last_sale', function($m) {
    return $m->refLink('Invoice')
        ->setOrder('date desc')
        ->setLimit(1)
        ->action('field', ['total_gross'], 'getOne');

});
```

The code above uses refLink and also creates expression, but it tweaks the action used.

Action Matrix

SQL actions apply the following:

- insert: init, mode
- update: init, mode, conditions, limit, order, hook
- delete: init, mode, conditions
- select: init, fields, conditions, limit, order, hook
- count: init, field, conditions, hook,
- field: init, field, conditions

- fx: init, field, conditions

CHAPTER 7

Fetching results

class Model

Model linked to a persistence in your "window" into DataSet and you get several ways which allow you to fetch the data. Apart from using ActiveRecord there are some other ways to fetch the data.

Iterate through model data

`Model::getIterator()`

Create your persistence object first then iterate it:

```
$db = \atk4\data\Persistence::connect($dsn);
$m = new Model_Client($db);

foreach($m as $id => $item) {
    echo $id.": ".$item['name']."\n";
}
```

You must be aware that $item will actually be same as $m and will point to the model. The model, however, will have the data loaded for you, so you can call methods for each iteration like this:

```
foreach($m as $item) {
    $item->sendReminder();
}
```

Additionally model will execute necessary after-load hooks that might trigger some other calculation or validations.

Note: changing query parameter during iteration will has no effect until you finish iterating.

Keeping models

If you wish to preserve the objects that you have loaded (not recommended as they will consume memory), you can do it like this:

```
$cat = [];

foreach(new Model_Category($db) as $id => $c) {
    $cat[$id] = clone $c;
}
```

Raw Data Fetching

`Model::rawIterator()`

If you do not care about the hooks and simply wish to get the data, you can fetch it:

```
foreach($m->rawIterator() as $row) {
    var_dump($row); // array
}
```

The $row will also contain value for "id" and it's up to you to find it yourself if you need it.

`Model::export()`

Will fetch and output array of hashes which will represent entirety of data-set. Similarly to other methods, this will have the data mapped into your fields for you and server-side expressions executed that are embedded in the query.

By default - 'only_fields' will be presented as well as system fields.

Fetching data through action

You can invoke and iterate action (particularly SQL) to fetch the data:

```
foreach($m->action('select') as $row) {
    var_dump($row); // array
}
```

This has the identical behavior to $m->rawIterator();

Comparison of various ways of fetching

- getIterator - action(select), [fetches row, set ID/Data, call afterLoad hook, yields model], unloads data
- rawIterator - action(select), [fetches row, yields row]
- export - action(select), fetches all rows, returns all rows

CHAPTER 8

Fields

Field represents a model *property* which we do refer to as field throughout Agile Data, to distinguish it from object properties. Fields inside a model normally have a corresponding instance of Field class.

See *Model::addField()* on how fields are added. By default, persistence sets the property _default_class_addField which should correspond to a field object that has enough capabilities for performing field-specific mapping into persistence-logic.

class `Field`

When no value is specified for a field, default value is used when inserting.

Valid types are: string, integer, boolean, datetime, date, time.

You can specify unsupported type. It will be untouched by Agile Data so you would have to implement your own handling of a new type.

Persistence implements two methods:

- `Persistence::typecastSaveRow()`
- `Persistence::typecastLoadRow()`

Those are responsible for converting PHP native types to persistence specific formats as defined in fields. Those methods will also change name of the field if needed (see Field::actual)

Specifies array containing all the possible options for the value. You can set only to one of the values (loosely typed comparison is used)

Set this to true if field property is mandatory and must be non-null in order for model to properly exist. Note that you can still set the NULL value to the field, but you won't be able to save it.

Modifying field that is read-only through set() methods (or array access) will result in exception. *Field_SQL_Expression* is read-only by default.

Specify name of the Table Row Field under which field will be persisted.

This property will point to *Join* object if field is associated with a joined table row.

System flag is intended for fields that are important to have inside hooks or some core logic of a model. System fields will always be appended to *Model::onlyFields*, however by default they will not appear on forms or grids (see `Model::isVisible`, `Model::isEditable`).

Adding condition on a field will also make it system.

Field will never be loaded or saved into persistence. You can use this flag for fields that physically are not located in the database, yet you want to see this field in beforeSave hooks.

This field will be loaded normally, but will not be saved in a database. Unlike "read_only" which has a similar effect, you can still change the value of this field. It will simply be ignored on save. You can create some logic in beforeSave hook to read this value.

This field contains certain arguments that may be needed by the UI layer to know if user should be allowed to edit this field.

Specify a callback that will be executed when the field is loaded and it is necessary to decode or do something else with loaded the value.

You can use this callback if you are storing data in some unusual format and need to convert it into PHP value. Format of callback is:

```
function ($value) {
    return str_rot13($value);
}
```

There are additional arguments in case you want to have a common callback:

```
$encrypt = function ($value, $key, $persistence) {

    // load encrypted data from SQL
    if ($persistence instanceof \atk4\data\Persistence_SQL) {
        return mcrypt_decrypt(MCRYPT_RIJNDAEL_128, $key->key, $value);
    }

    return $value;
}
```

Note that if you use a call-back this will by-pass normal field typecasting.

See Advanced::EncryptedField for full example.

Same as loadCallback property but will be executed when saving data. Arguments are still the same:

```
function ($value) {
    return str_rot13($value);
}
```

There are additional arguments in case you want to have a common callback:

```
$decrypt = function ($value, $key, $persistence) {

    // load encrypted data from SQL
    if ($persistence instanceof \atk4\data\Persistence_SQL) {
        return mcrypt_encrypt(MCRYPT_RIJNDAEL_128, $key->key, $value);
    }

    return $value;
}
```

See Advanced::EncryptedField for full example.

`Field::set()`

Set the value of the field. Same as $model->set($field_name, $value);

`Field::get()`

Get the value of the field. Same as $model->get($field_name, $value);

UI Presentation

Agile Data does not deal directly with formatting your data for the user. There may be various items to consider, for instance the same date can be presented in a short or long format for the user.

The UI framework such as Agile Toolkit can make use of the `Field::ui` property to allow user to define default formats or input parsing rules, but Agile Data does not regulate the `Field::ui` property and different UI frameworks may use it differently.

`Field::isEditable()`

Returns true if UI should render this field as editable and include inside forms by default.

`Field::isVisible()`

Returns true if UI should render this field in Grid and other read_only display views by default.

`Field::isHidden()`

Returns true if UI should not render this field in views.

CHAPTER 9

Conditions and DataSet

class Model

When model is associated with the database, you can specify a default table either explicitly or through a $table property inside a model:

```
$m = new Model_User($db, 'user');
$m->load(1);
echo $m['gender'];   // "M"
```

Following this line, you can load ANY record from the table. It's possible to narrow down set of "loadable" records by introducing a condition:

```
$m = new Model_User($db, 'user');
$m->addCondition('gender','F');
$m->load(1);    // exception, user with ID=1 is M
```

Conditions serve important role and must be used to intelligently restrict logically accessible data for a model before you attempt the loading.

Basic Usage

Model::**addCondition**(*$field*, *$operator* = *null*, *$value* = *null*)

There are many ways to execute addCondition. The most basic one that will be supported by all the drivers consists of 2 arguments or if operator is '=':

```
$m->addCondition('gender', 'F');         // or
$m->addCondition('gender', '=', 'F');
```

Once you add a condition, you can't get rid of it, so if you want to preserve the state of your model, you need to use clone:

```
$m = new Model_User($db, 'user');
$girls = (clone $m)->addCondition('gender','F');

$m->load(1);        // success
$girls->load(1);    // exception
```

Operations

Most database drivers will support the following additional operations:

```
>, <, >=, <=, !=, in, not in
```

The operation must be specified as second argument:

```
$m = new Model_User($db, 'user');
$girls = (clone $m)->addCondition('gender', 'F');
$not_girls = (clone $m)->addCondition('gender', '!=', 'F');
```

When you use 'in' or 'not in' you should pass value as array:

```
$m = new Model_User($db, 'user');
$girls_or_boys = (clone $m)->addCondition('gender', 'in', ['F', 'M']);
```

Multiple Conditions

You can set multiple conditions on the same field even if they are contradicting:

```
$m = new Model_User($db, 'user');
$noone = (clone $m)
    ->addCondition('gender', 'F')
    ->addCondition('gender', 'M');
```

Normally, however, you would use a different fields:

```
$m = new Model_User($db, 'user');
$girl_sue = (clone $m)
    ->addCondition('gender', 'F')
    ->addCondition('name', 'Sue');
```

You can have as many conditions as you like.

Adding OR Conditions

In Agile Data all conditions are additive. This is done for security - no matter what condition you are adding, it will not allow you to circumvent previously added condition.

You can, however, add condition that contains multiple clauses joined with OR operator:

```
$m->addCondition([
    ['name', 'John'],
    ['surname', 'Smith']
]);
```

This will add condition that will match against records with either name=John OR surname=Smith. If you are building multiple conditions against the same field, you can use this format:

```
$m->addCondition('name', ['John', 'Joe']);
```

For all other cases you can implement them with *Model::expr*:

```
$m->addCondition($m->expr("(day([birth_date]) = day([registration_date]) or␣
↪day([birth_date]) = [])", 10));
```

This rather unusual condition will show user records who have registered on same date when they were born OR if they were born on 10th. (This is really silly condition, please don't judge, if you have a better example, I'd love to hear).

Defining your classes

Although I have used in-line addition of the arguments, normally you would want to set those conditions inside the init() method of your model:

```
class Model_Girl extends Model_User
{
    function init()
    {
        parent::init();

        $this->addCondition('gender', 'F');
    }
}
```

Note that the field 'gender' should be defined inside Model_User::init().

Vendor-dependent logic

There are many other ways to set conditions, but you must always check if they are supported by the driver that you are using.

Field Matching

Supported by: SQL (planned for Array, Mongo)

Usage:

```
$m->addCondition('name', $m->getElement('surname'));
```

Will perform a match between two fields.

Expression Matching

Supported by: SQL (planned for Array)

Usage:

```
$m->addCondition($m->expr('[name] > [surname]');
```

Allow you to define an arbitrary expression to be used with fields. Values inside [blah] should correspond to field names.

SQL Expression Matching

Model::**expr**(*$expression*, *$arguments* =[])

Basically is a wrapper to create DSQL Expression, however this will find any usage of identifiers inside the template that do not have a corresponding value inside $arguments and replace it with the field:

```
$m->expr('[age] > 20'); // same as
$m->expr('[age] > 20', ['age'=>$m->getElement('age')); // same as
```

Supported by: SQL

Usage:

```
$m->addCondition($m->expr('[age] between [min_age] and [max_age]'));
```

Allow you to define an arbitrary expression using SQL language.

Custom Parameters in Expressions

Supported by: SQL

Usage:

```
$m->addCondition(
    $m->expr('[age] between [min_age] and [max_age]'),
    ['min_age'=>10, 'max_age'=>30]
);
```

Allow you to pass parameters into expressions. Those can be nested and consist of objects as well as actions:

```
$m->addCondition(
    $m->expr('[age] between [min_age] and [max_age]'),
    [
        'min_age'=>$m->action('min', ['age']),
        'max_age'=>$m->expr('(20 + [])', [20])
    ]
);
```

This will result in the following condition:

```
WHERE
    `age` between
        (select min(`age`) from `user`)
        and
        (20 + :a)
```

where the other 20 is passed through parameter. Refer to http://dsql.readthedocs.io/en/develop/expressions.html for full documentation on expressions.

Expression as first argument

Supported by: SQL, (Planned: Array, Mongo)

The $field of addCondition() can be passed as either an expression or any object implementing atk4dsqlExpressionable interface. Same logic applies to the $value:

```
$m->addCondition($m->getElement('name'), '!=', $this->getElement('surname'));
```

Using withID

`Model::withID(`*$id*`)`

This method is similar to load($id) but instead of loading the specified record, it sets condition for ID to match. Technically that saves you one query if you do not need actual record by are only looking to traverse:

```
$u = new Model_User($db);
$books = $u->withID(20)->ref('Books');
```

CHAPTER 10

SQL Extensions

Databases that support SQL language can use *Persistence_SQL*. This driver will format queries to the database using SQL language.

In addition to normal operations you can extend and customise various queries.

Default Model Classes

When using Persistence_SQL model building will use different classes for fields, expressions, joins etc:

- addField - *Field_SQL* (field can be used as part of DSQL Expression)
- hasOne - *Reference_SQL_One* (allow importing fields)
- addExpression - *Field_SQL_Expression* (define expression through DSQL)
- join - *Join_SQL* (join tables query-time)

SQL Field

class `Field_SQL`

property `Field_SQL::$`**`actual`**

Persistence_SQL supports field name mapping. Your field could have different column name in your schema:

```
$this->addField('name', ['actual'=>'first_name']);
```

This will apply to load / save operations as well as query mapping.

`Field_SQL::`**`getDSQLExpression`**`()`

SQL Fields can be used inside other SQL expressions:

```
$q = new \atk4\dsql\Expression('[age] + [birth_year]', [
        'age'        => $m->getElement('age'),
        'birth_year' => $m->getElement('birth_year'),
    ]);
```

SQL Reference

class Reference_SQL_One
Extends *Reference_One*

Reference_SQL_One::**addField**()
Allows importing field from a referenced model:

```
$model->hasOne('country_id', new Country())
    ->addField('country_name', 'name');
```

Second argument could be array containing additional settings for the field:

```
$model->hasOne('account_id', new Account())
    ->addField('account_balance', ['balance', 'type'=>'money']);
```

Returns new field object.

Reference_SQL_One::**addFields**()
Allows importing multiple fields:

```
$model->hasOne('country_id', new Country())
    ->addFields(['country_name', 'country_code']);
```

You can specify defaults to be applied on all fields:

```
$model->hasOne('account_id', new Account())
    ->addFields([
        'opening_balance',
        'balance'
    ], ['type'=>'money']);
```

You can also specify aliases:

```
$model->hasOne('account_id', new Account())
    ->addFields([
        'opening_balance',
        'account_balance'=>'balance'
    ], ['type'=>'money']);
```

If you need to pass more details to individual field, you can also use sub-array:

```
$model->hasOne('account_id', new Account())
    ->addFields([
    [
        ['opening_balance', 'caption'=>'The Opening Balance'],
        'account_balance'=>'balance'
    ], ['type'=>'money']);
```

Returns $this.

Reference_SQL_One::**ref**()
While similar to *Reference_One::ref* this implementation implements deep traversal:

```
$country_model = $customer_model->addCondition('is_vip', true)
    ->ref('country_id');           // $model was not loaded!
```

Reference_SQL_One::**refLink**()
Creates a model for related entity with applied condition referencing field of a current model through SQL expression rather then value. This is usable if you are creating sub-queries.

Reference_SQL_One::**addTitle**()
Similar to addField, but will import "title" field and will come up with good name for it:

```
$model->hasOne('country_id', new Country())
    ->addTitle();

// creates 'country' field as sub-query for country.name
```

You may pass defaults:

```
$model->hasOne('country_id', new Country())
    ->addTitle(['caption'=>'Country Name']);
```

Returns new field object.

Reference_SQL_One::**withTitle**()
Similar to addTitle, but returns $this.

Expressions

class Field_SQL_Expression
Extends *Field_SQL*

Expression will map into the SQL code, but will perform as read-only field otherwise.

property Field_SQL_Expression::$**expr**
Stores expression that you define through DSQL expression:

```
$model->addExpression('age', 'year(now())-[birth_year]');
// tag [birth_year] will be automatically replaced by respective model field
```

Field_SQL_Expression::**getDSQLExpression**()
SQL Expressions can be used inside other SQL expressions:

```
$model->addExpression('can_buy_alcohol', ['if([age] > 25, 1, 0)', 'type'=>'boolean
↪']);
```

Adding expressions to model will make it automatically reload itself after save as default behaviour, see Model::reload_after_save.

Transactions

class Persistence_SQL

Persistence_SQL::**atomic**()

This method allows you to execute code within a 'START TRANSACTION / COMMIT' block:

```
class Invoice {

    function applyPayment(Payment $p) {

        $this->persistence->atomic(function() use ($p) {

            $this['paid'] = true;
            $this->save();

            $p['applied'] = true;
            $p->save();

        });

    }
}
```

Callback format of this method allows a more intuitive syntax and nested execution of various blocks. If any exception is raised within the block, then transaction will be automatically rolled back. The return of atomic() is same as return of user-defined callback.

Custom Expressions

`Persistence_SQL::expr()`
: This method is also injected into the model, that is associated with Persistence_SQL so the most convenient way to use this method is by calling *$model->expr('foo')*.

This method is quite similar to atk4dsqlQuery::expr() method explained here: http://dsql.readthedocs.io/en/stable/expressions.html

There is, however, one difference. Expression class requires all named arguments to be specified. Use of Model::expr() allows you to specify field names and those field expressions will be automatically substituted. Here is long / short format:

```
$q = new \atk4\dsql\Expression('[age] + [birth_year]', ['age'=>$m->getElement('age'),
↪'birth_year'=>$m->getElement('birth_year')]);

// identical to

$q = $m->expr('[age] + [birth_year]');
```

This method is automatically used by *Field_SQL_Expression*.

Actions

The most basic action you can use with SQL persistence is 'select':

```
$action = $model->action('select');
```

Action is implemented by DSQL library, that is further documented at http://dsql.readthedocs.io (See section Queries).

Action: select

This action returns a basic select query. You may pass one argument - array containing list of fields:

```
$action = $model->action('select', ['name', 'surname']);
```

Passing false will not include any fields into select (so that you can include them yourself):

```
$action = $model->action('select', [false]);
$action->field('count(*)', 'c);
```

Action: insert

Will prepare query for performing insert of a new record.

Action: update, delete

Will prepare query for performing update or delete of records. Applies conditions set.

Action: count

Returns query for *count(*)*:

```
$action = $model->action('count');
$cnt = $action->getOne();
```

You can also specify alias:

```
$action = $model->action('count', ['alias'=>'cc']);
$data = $action->getRow();
$cnt = $data['cc'];
```

Action: field

Get query for a specific field:

```
$action = $model->action('field', ['age']);
$age = $action->limit(1)->getOne();
```

You can also specify alias:

```
$action = $model->action('field', ['age', 'alias'=>'the_age']]);
$age = $action->limit(1)->getRow()['the_age'];
```

Action: fx

Executes single-argument SQL function on field:

```
$action = $model->action('fx', ['avg', 'age']);
$avg_age = $action->getOne();
```

This method also supports alias. Use of alias is handy if you are using those actions as part of other query (e.g. UNION)

CHAPTER 11

References

class Model

ref($link, $details = []);

Models can relate one to another. The logic of traversing references, however, is slightly different to the traditional ORM implementation, because in Agile Data traversing also imposes *Conditions and DataSet*

There are two basic types of references: hasOne() and hasMany(), but it's also possible to add other reference types. The basic ones are really easy to use:

```
$m = new Model_User($db, 'user');
$m->hasMany('Orders', new Model_Order());
$m->load(13);

$orders_for_user_13 = $m->ref('Orders');
```

As mentioned - $orders_for_user_13 will have it's DataSet automatically adjusted so that you could only access orders for the user with ID=13. The following is also possible:

```
$m = new Model_User($db, 'user');
$m->hasMany('Orders', new Model_Order());
$m->addCondition('is_vip', true);

$orders_for_vips = $m->ref('Orders');
$orders_for_vips->loadAny();
```

Condition on the base model will be carried over to the orders and you will only be able to access orders that belong to VIP users. The query for loading order will look like this:

```
select * from order where user_id in (
    select id from user where is_vip = 1
) limit 1
```

Argument $defaults will be passed to the new model that will be used to create referenced model. This will not work if you have specified reference as existing model that has a persistence set. (See Reference::getModel())

Persistence

Agile Data supports traversal between persistences. The code above does not explicitly assign database to Model_Order. But what if destination model does not reside inside the same database?

You can specify it like this:

```
$m = new Model_User($db_array_cache, 'user');
$m->hasMany('Orders', new Model_Order($db_sql));
$m->addCondition('is_vip', true);

$orders_for_vips = $m->ref('Orders');
```

Now that a different databases are used, the queries can no longer be joined so Agile Data will carry over list of IDs instead:

```
$ids = select id from user where is_vip = 1
select * from order where user_id in ($ids)
```

Since we are using `$db_array_cache`, then field values will actually be retrieved from memory.

Note: This is not implemented as of 1.1.0, see https://github.com/atk4/data/issues/158

Safety and Performance

When using ref() on hasMany reference, it will always return a fresh clone of the model. You can perform actions on the clone and next time you execute ref() you will get a fresh copy.

If you are worried about performance you can keep 2 models in memory:

```
$order = new Order($db);
$client = $order->refModel('client_id');

foreach($order as $o) {
    $client->load($o['client_id']);
}
```

Warning: This code is seriously flawed and is called "N+1 Problem". Agile Data discourages you from using this and instead offers you many other tools: field importing, model joins, field actions and refLink().

hasMany Reference

`hasMany($link, $model);`

There are several ways how to link models with hasMany:

```
$m->hasMany('Orders', new Model_Order());  // using object

$m->hasMany('Order', function($m, $r) {    // using callback
    return new Model_Order();
});
```

```
$m->hasMany('Order');                       // will use factory new Model_Order
```

Dealing with many-to-many references

It is possible to perform reference through an 3rd party table:

```
$i = new Model_Invoice();
$p = new Model_Payment();

// table invoice_payment has 'invoice_id', 'payment_id' and 'amount_allocated'

$p
    ->join('invoice_payment.payment_id')
    ->addFields(['amount_allocated','invoice_id']);

$i->hasMany('Payments', $p);
```

Now you can fetch all the payments associated with the invoice through:

```
$payments_for_invoice_1 = $i->load(1)->ref('Payments');
```

Dealing with NON-ID fields

Sometimes you have to use non-ID references. For example, we might have two models describing list of currencies and for each currency we might have historic rates available. Both models will relate through `currency.code = exchange.currency_code`:

```
$c = new Model_Currency();
$e = new Model_ExchangeRate();

$c->hasMany('Exchanges', [$e, 'their_field'=>'currency_code', 'our_field'=>'code']);

$c->addCondition('is_convertable',true);
$e = $c->ref('Exchanges');
```

This will produce the following query:

```
select * from exchange
where currency_code in
    (select code form currency where is_convertable=1)
```

Add Aggregate Fields

Reference hasMany makes it a little simpler for you to define an aggregate fields:

```
$u = new Model_User($db_array_cache, 'user');

$u->hasMany('Orders', new Model_Order())
    ->addField('amount', ['aggregate'=>'sum']);
```

It's important to define aggregation functions here. This will add another field inside `$m` that will correspond to the sum of all the orders. Here is another example:

```
$u->hasMany('PaidOrders', (new Model_Order())->addCondition('is_paid', true))
    ->addField('paid_amount', ['aggregate'=>'sum', 'field'=>'amount']);
```

You can also define multiple fields, although you must remember that this will keep making your query bigger and bigger:

```
$invoice->hasMany('Invoice_Line', new Model_Invoice_Line())
    ->addFields([
        ['total_vat', 'aggregate'=>'sum'],
        ['total_net', 'aggregate'=>'sum'],
        ['total_gross', 'aggregate'=>'sum'],
    ]);
```

Important: Imported fields will preserve format of the field the reference. In the example, if 'Invoice_line' field total_vat has type *money* then it will also be used for a sum. Aggregate fields are always declared read-only, and if you try to change them, you will receive exception.

hasMany / refLink / refModel

`Model::refLink(`*$link*`)`

Normally ref() will return a usable model back to you, however if you use refLink then the conditioning will be done differently. refLink is useful when defining sub-queries:

```
$m = new Model_User($db_array_cache, 'user');
$m->hasMany('Orders', new Model_Order($db_sql));
$m->addCondition('is_vip', true);

$sum = $m->refLink('Orders')->action('fx0', ['sum', 'amount']);
$m->addExpression('sum_amount')->set($sum);
```

The refLink would define a condition on a query like this:

```
select * from `order` where user_id = `user`.id
```

And it will not be viable on its own, however if you use it inside a sub-query, then it now makes sense for generating expression:

```
select
    (select sum(amount) from `order` where user_id = `user`.id) sum_amount
from user
where is_vip = 1
```

`Model::refModel(`*$link*`)`

There are many situations when you need to get referenced model instead of reference itself. In such case refModel() comes in as handy shortcut of doing *$model->refLink($link)->getModel()*.

hasOne reference

Model::**hasOne**(*$link*, *$model*)
$model can be an array containing options: [$model, ...]

This reference allows you to attach a related model to a foreign key:

```
$o = new Model_Order($db, 'order');
$u = new Model_User($db, 'user');

$o->hasOne('user_id', $u);
```

This reference is similar to hasMany, but it does behave slightly different. Also this reference will define a system new field `user_id` if you haven't done so already.

Traversing loaded model

If your `$o` model is loaded, then traversing into user will also load the user, because we specifically know the ID of that user. No conditions will be set:

```
echo $o->load(3)->ref('user_id')['name'];    // will show name of the user, of order #3
```

Traversing DataSet

If your model is not loaded then using ref() will traverse by conditioning DataSet of the user model:

```
$o->unload(); // just to be sure!
$o->addCondition('status', 'failed');
$u = $o->ref('user_id');


$u->loadAny();  // will load some user who has at least one failed order
```

The important point here is that no additional queries are generated in the process and the loadAny() will look like this:

```
select * from user where id in
    (select user_id from order where status = 'failed')
```

By passing options to hasOne() you can also differentiate field name:

```
$o->addField('user_id');
$o->hasOne('User', [$u, 'our_field'=>'user_id']);

$o->load(1)->ref('User')['name'];
```

You can also use `their_field` if you need non-id matching (see example above for hasMany()).

Importing Fields

You can import some fields from related model. For example if you have list of invoices, and each invoice contains "currency_id", but in order to get the currency name you need another table, you can use this syntax to easily import

the field:

```
$i = new Model_Invoice($db)
$c = new Model_Currency($db);

$i->hasOne('currency_id', $c)
    ->addField('currency_name', 'name');
```

This code also resolves problem with a duplicate 'name' field. Since you might have a 'name' field inside 'Invoice' already, you can name the field 'currency_name' which will reference 'name' field inside Currency. You can also import multiple fields but keep in mind that this may make your query much longer. The argument is associative array and if key is specified, then the field will be renamed, just as we did above:

```
$u = new Model_User($db)
$a = new Model_Address($db);

$u->hasOne('address_id', $a)
    ->addFields([
        'address_1',
        'address_2',
        'address_3',
        'address_notes'=>['notes', 'type'=>'text']
    ]);
```

Above, all `address_` fields are copied with the same name, however field 'notes' from Address model will be called 'address_notes' inside user model.

Important: When importing fields, they will preserve type, e.g. if you are importing 'date' then the type of your imported field will also be date. Imported fields are also marked as "read-only" and attempt to change them will result in exception.

Importing hasOne Title

When you are using hasOne() in most cases the referenced object will be addressed through "ID" but will have a human-readable field as well. In the example above *Model_Currency* has a title field called *name*. Agile Data provides you an easier way how to define currency title:

```
$i = new Invoice($db)

$i->hasOne('currency_id', new Currency())
    ->addTitle();
```

This would create 'currency' field containing name of the curerncy:

```
$i->load(20);

echo "Currency for invoice 20 is ".$i['currency'];    // EUR
```

Unlike addField() which creates fields read-only, title field can in fact be modified:

```
$i['currency'] = 'GBP';
$i->save();

// will update $i['currency_id'] to the corresponding ID for currency with name GBP.
```

This behaviour is awesome when you are importing large amounts of data, because the lookup for the currency_id is entirely done in a database.

By default name of the field will be calculated by removing "_id" from the end of hasOne field, but to override this, you can specify name of the title field explicitly:

```
$i->hasOne('currency_id', new Currency())
    ->addTitle(['field'=>'currency_name']);
```

User-defined Reference

`Model::addRef(`*$link*, *$callback*`)`

Sometimes you would want to have a different type of relation between models, so with *addRef* you can define whatever reference you want:

```
$m->addRef('Archive', function($m) {
    return $m->newInstance(null, ['table' => $m->table.'_archive']);
});
```

The above example will work for a table structure where a main table *user* is shadowed by a archive table *user_archive*. Structure of both tables are same, and if you wish to look into an archive of a User you would do:

```
$user->ref('Archive');
```

Note that you can create one-to-many or many-to-one relations, by using your custom logic. No condition will be applied by default so it's all up to you:

```
$m->addRef('Archive', function($m) {
    $archive = $m->newInstance(null, ['table' => $m->table.'_archive']);

    $m->addField('original_id', ['type' => 'int']);

    if ($m->loaded()) {
        $archive->addCondition('original_id', $m->id);
        // only show record of currently loaded record
    }
});
```

Reference Discovery

You can call `Model::getRefs()` to fetch all the references of a model:

```
$refs = $model->getRefs();
$ref = $refs['owner_id'];
```

or if you know the reference you'd like to fetch, you can use `Model::getRef()`:

```
$ref = $model->getRef('owner_id');
```

While `Model::ref()` returns a related model, `Model::getRef()` gives you the reference object itself so that you could perform some changes on it, such as import more fields with *`Model::addField()`*.

Or you can use *`Model::refModel()`* which will simply return referenced model and you can do fancy things with it.

$ref_model = $model->refModel('owner_id');

You can also use `Model::hasRef()` to check if particular reference exists in model:

```
$ref = $model->hasRef('owner_id');
```

Deep traversal

When operating with data-sets you can define references that use deep traversal:

```
echo $o->load(1)->ref('user_id')->ref('address_id')['address_1'];
```

The above example will actually perform 3 load operations, because as I have explained above, `Model::ref()` loads related model when called on a loaded model. To perform a single query instead, you can use:

```
echo $o->withID(1)->ref('user_id')->ref('address_id')->loadAny()['address_1'];
```

Here `withID()` will only set a condition without actually loading the record and traversal will encapsulate sub-queries resulting in a query like this:

```
select * from address where id in
    (select address_id from user where id in
        (select user_id from order where id=1 ))
```

Reference Aliases

When related entity relies on the same table it is possible to run into problem when SQL is confused about which table to use.

```
select name, (select name from item where item.parent_id = item.id) parent_name from
↪item
```

To avoid this problem Agile Data will automatically alias tables in sub-queries. Here is how it works:

```
$item->hasMany('parent_item_id', new Model_Item())
    ->addField('parent', 'name');
```

When generating expression for 'parent', the sub-query will use alias `pi` consisting of first letters in 'parent_item_id'. (except _id). You can actually specify a custom table alias if you want:

```
$item->hasMany('parent_item_id', [new Model_Item(), 'table_alias'=>'mypi'])
    ->addField('parent', 'name');
```

Additionally you can pass table_alias as second argument into `Model::ref()` or *`Model::refLink()`*. This can help you in creating a recursive models that relate to itself. Here is example:

```
class Model_Item3 extends \atk4\data\Model {
    public $table='item';
    function init() {
        parent::init();

        $m = new Model_Item3();

        $this->addField('name');
        $this->addField('age');
```

```
        $i2 = $this->join('item2.item_id');
        $i2->hasOne('parent_item_id', [$m, 'table_alias'=>'parent'])
            ->addTitle();

        $this->hasMany('Child', [$m, 'their_field'=>'parent_item_id', 'table_alias'=>
↪'child'])
            ->addField('child_age',['aggregate'=>'sum', 'field'=>'age']);
    }
}
```

Loading model like that can produce a pretty sophisticated query:

```
select
    `pp`.`id`,`pp`.`name`,`pp`.`age`,`pp_i`.`parent_item_id`,
    (select `parent`.`name`
     from `item` `parent`
     left join `item2` as `parent_i` on `parent_i`.`item_id` = `parent`.`id`
     where `parent`.`id` = `pp_i`.`parent_item_id`
     ) `parent_item`,
    (select sum(`child`.`age`) from `item` `child`
     left join `item2` as `child_i` on `child_i`.`item_id` = `child`.`id`
     where `child_i`.`parent_item_id` = `pp`.`id`
    ) `child_age`,`pp`.`id` `_i`
from `item` `pp`left join `item2` as `pp_i` on `pp_i`.`item_id` = `pp`.`id`
```

Various ways to specify options

When calling *hasOne()->addFields()* there are various ways to pass options:

- *addFields(['name', 'dob'])* - no options are passed, use defaults. Note that reference will not fetch the type of foreign field due to performance consideration.
- *addFields(['first_name' => 'name'])* - this indicates aliasing. Field *name* will be added as *first_name*.
- *addFields([['dob', 'type'=>'date']])* - wrap inside array to pass options to field
- *addFields(['the_date' => ['dob', 'type'=>'date']])* - combination of aliasing and options
- *addFields(['dob', 'dod'], ['type'=>'date'])* - passing defaults for multiple fields

References with New Records

Agile Data takes extra care to help you link your new records with new related entities. Consider the following two models:

```
class Model_User extends \atk4\data\Model {
    public $table = 'user';
    function init() {
        parent::init();
        $this->addField('name');

        $this->hasOne('contact_id', new Model_Contact());
    }
}

class Model_Contact extends \atk4\data\Model {
```

```
    public $table = 'contact';
    function init() {
        parent::init();

        $this->addField('address');
    }
}
```

This is a classic one to one reference, but let's look what happens when you are working with a new model:

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

$m['name'] = 'John';
$m->save();
```

In this scenario, a new record will be added into 'user' with 'contact_id' equal to null. The next example will traverse into the contact to set it up:

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

$m['name'] = 'John';
$m->ref('address_id')->save(['address'=>'street']);
$m->save();
```

When entity which you have referenced through ref() is saved, it will automatically populate $m['contact_id'] field and the final $m->save() will also store the reference.

ID setting is implemented through a basic hook. Related model will have afterSave hook, which will update address_id field of the $m.

Reference Classes

References are implemented through several classes:

class `Reference_One`
Defines generic reference, that is typically created by *`Model::addRef`*

property `Reference_One::$table_alias`
Alias for related table. Because multiple references can point to the same table, ability to have unique alias is pretty good.

You don't have to change this property, it is generated automatically.

property `Reference_One::$link`
What should we pass into owner->ref() to get through to this reference. Each reference has a unique identifier, although it's stored in Model's elements as '#ref-xx'.

property `Reference_One::$model`
May store reference to related model, depending on implementation.

property `Reference_One::$our_field`
This is an optional property which can be used by your implementation to store field-level relationship based on a common field matching.

property `Reference_One::$their_filed`
This is an optional property which can be used by your implementation to store field-level relationship based on a common field matching.

`Reference_One::getModel()`
: Returns referenced model without conditions.

`Reference_One::ref()`
: Returns referenced model WITH conditions. (if possible)

`Reference_One::guessFieldType()`
: This method implementation is removed due to performance but may be reconsidered. Attempts to initialize related model to find out more about the field that is being referenced during the "definition time".

  Normally this would happen only during query time and if the field is included into query.

CHAPTER 12

Expressions

class Model

You already know that you can define fields inside your Model with addField. While a regular field maps to physical field inside your database, sometimes you want to do something different - execute expression or function inside SQL and use result as an output.

Expressions solve this problem by adding a read-only field to your model that corresponds to an expression:

addExpression($link, $model);

Example will calculate "total_gross" by adding up values for "net" and "vat":

```
$m = new Model_Invoice($db);
$m->addFields(['total_net', 'total_vat']);

$m->addExpression('total_gross', '[total_net]+[total_vat]');
$m->load(1);

echo $m['total_gross'];
```

The query using during load() will look like this:

```
select
    `id`,`total_net`,`total_vat`,
    (`total_net`+`total_vat`) `total_gross`
from `invoice`',
```

Defining Expression

The simplest format to define expression is by simply passing a string. The argument is executed through Model::expr() which automatically substitutes values for the other fields including other expressions.

There are other ways how you can specify expression:

```
$m->addExpression('total_gross',
    $m->expr('[total_net]+[total_vat] + [fee]', ['fee'=>$fee])
);
```

This format allow you to supply additional parameters inside expression. You should always use parameters instead of appending values inside your expression string (for safety)

You can also use expressions to pass a select action for a specific field:

No-table Model Expression

Agile Data allows you to define a model without table. While this may have no purpose initially, it does come in handy in some cases, when you need to unite a few statistical queries. Let's start by looking a at a very basic example:

```
$m = new Model($db, false);
$m->addExpression('now', 'now()');
$m->loadAny();
echo $m['now'];
```

In this example the query will look like this:

```
select (1) `id`, (now()) `now` limit 1
```

so that `$m->id` will always be 1 which will make it a model that you can actually use consistently throughout the system. The real benefit from this can be gained when you need to pull various statistical values from your database at once:

```
$m = new Model($db, false);
$m->addExpression('total_orders', (new Model_Order($db))->action('count'));
$m->addExpression('total_payments', (new Model_Payment($db))->action('count'));
$m->addExpression('total_received', (new Model_Payment($db))->action('fx0', ['sum',
↪'amount']));

$data = $m->loadAny()->get();
```

Of course you can also use a DSQL for this:

```
$q = $db->dsql();
$q->field(new Model_Order($db)->action('count'), 'total_orders');
$q->field(new Model_Payment($db)->action('count'), 'total_orders');
$q->field(new Model_Payment($db)->action('fx0', ['sum', 'amount']), 'total_received');
$data = $q->getRow();
```

You can decide for yourself based on circumstances.

Expression Callback

You can use a callback method when defining expression:

```
$m->addExpression('total_gross', function($m, $q) {
    return '[total_net]+[total_vat]';
});
```

Model Reloading after Save

When you add SQL Expressions into your model, that means that some of the fields might be out of sync and you might need your SQL to recalculate those expressions.

To simplify your life, Agile Data implements smart model reloading. Consider the following model:

```
class Model_Math extends \atk4\data\Model
{
    public $table = 'math';
    function init()
    {
        parent::init();

        $this->addFields(['a', 'b']);

        $this->addExpression('sum', '[a]+[b]');
    }
}

$m = new Model_Math($db);
$m['a'] = 4;
$m['b'] = 6;

$m->save();

echo $m['sum'];
```

When $m->save() is executed, Agile Data will perform reloading of the model. This is to ensure that expression 'sum' would be re-calculated for the values of 4 and 6 so the final line will output a desired result - 10;

Reload after save will only be executed if you have defined any expressions inside your model, however you can affect this behavior:

```
$m = new Model_Math($db, ['reload_after_save' => false]);
$m['a'] = 4;
$m['b'] = 6;

$m->save();

echo $m['sum'];    // outputs null

$m->reload();
echo $m['sum'];    // outputs 10
```

Now it requires an explicit reload for your model to fetch the result. There is another scenario when your database defines default fields:

```
alter table math change b b int default 10;
```

Then try the following code:

```
class Model_Math extends \atk4\data\Model
{
    public $table = 'math';
    function init()
    {
        parent::init();
```

```
        $this->addFields(['a', 'b']);
    }
}

$m = new Model_Math($db);
$m['a'] = 4;

$m->save();

echo $m['a']+$m['b'];
```

This will output 4, because model didn't reload itself due to lack of any expressions. This time you can explicitly enable reload after save:

```
$m = new Model_Math($db, ['reload_after_save' => true]);
$m['a'] = 4;

$m->save();

echo $m['a']+$m['b']; // outputs 14
```

Note: If your model is using reload_after_save, but you wish to insert data without additional query - use *Model::insert()* or *Model::import()*.

CHAPTER 13

Model from multiple joined table

class `Join`

Sometimes model logically contains information that is stored in various places in the database. Your database may want to split up logical information into tables for various reasons, such as to avoid repetition or to better optimize indexes.

Join Basics

Agile Data allows you to map multiple table fields into a single business model by using joins:

```
$user->addField('username');
$j_contact = $user->join('contact');
$j_contact->addField('address');
$j_contact->addField('county');
$j_contact->hasOne('Country');
```

This code will load data from two tables simultaneously and if you do change any of those fields they will be update in their respective tables. With SQL the load query would look like this:

```
select
    u.username, c.address, c.county, c.country_id
    (select name from country where country.id=c.country_id) country
from user u
join contact c on c.id=u.contact_id
where u.id = $id
```

If driver is unable to query both tables simultaneously, then it will load one record first, then load other record and will collect fields together:

```
$user_data = $user->find($id);
$contact_data = $contact->find($user_data['contact_id']);
```

When saving the record, Joins will automatically record data correctly:

```
insert into contact (address, county, country_id) values ($, $, $);
@join_c = last_insert_id();
insert into user (username, contact_id) values ($, @join_c)
```

Strong and Weak joins

When you are joining tables, then by default a strong join is used. That means that both records are mandatory and when adding records, they will both be added and linked.

Weak join is used if you do not really want to modify the other table. For example it can be used to pull country information based on user.country_id but you wouldn't want that adding a new user would create a new country:

```
$user->addField('username');
$user->addField('country_id');
$j_country = $user->weakJoin('country', ['prefix'=>'country_']);
$j_country->addField('code');
$j_country->addField('name');
$j_country->addField('default_currency', ['prefix'=>false]);
```

After this you will have the following fields in your model:

- username
- country_id
- country_code [read_only]
- country_name [read_only]
- default_currency [read_only]

`Join::imortModel()`

You can achieve a similar functionality with hasOne reference, but with weak join you can pull multiple fields into your model. Finally you can even join using existing models:

```
$user->addField('username');
$user->addField('country_id');
$user->weakJoin('country')->importModel('Country');
```

This will automatically import fields, expressions, references and conditions from 'Country' model into $user model and will also re-map field names in process.

`Join::weakJoinModel()`

To save you some time with weakJoin() and importModel(), if you wish to simply import another model fields, you can actually use this syntax:

```
$user->weakJoinModel('Country', ['code', 'name', 'default_currency']);
```

When joining model like that, all the fields will be prefixed automatically using Country::$table property.

Join relationship definitions

When defining joins, you need to outline two fields that must match. In our earlier examples, we the master table was "user" that contained reference to "contact". The condition would look like this `user.contact_id=contact.id`. In some cases, however, a relation should be reversed:

```
$j_contact = $user->join('contact.user_id');
```

This will result in the following join condition: `user.id=contact.user_id`. The first argument to join defines both the table that we need to join and can optionally define the field in the foreign table. If field is set, we will assume that it's a reverse join.

Reverse joins are saved in the opposite order - primary table will be saved first and when id of a primary table is known, foreign table record is stored and ID is supplied. You can pass option 'master_field' to the join() which will specify which field to be used for matching. By default the field is calculated like this: foreign_table.'_id'. Here is usage example:

```
$user->addField('username');
$j_cc = $user->join('credit_card', [
    'prefix'=>'cc_',
    'master_field'=>'default_credit_card_id'
]);
$j_cc->addField('number');  // creates cc_number
$j_cc->addField('name');    // creates cc_name
```

Master field can also be specified as an object of a Field class.

There are more options that you can pass inside join(), but those are vendor-specific and you'll have to look into documentation for sqlJoin and mongoJoin respectfully.

Method Proxying

Once your join is defined, you can call several methods on the join objects, that will create fields, other joins or expressions but those would be associated with a foreign table.

`Join::addField()`
: same as *`Model::addField`* but associates field with foreign table.

`Join::join()`
: same as `Model::join` but links new table with this foreign table.

`Join::weakJoin()`
: same as `Model::weakJoin` but links new table with this foreign table.

`Join::hasOne()`
: same as *`Model::hasOne`* but reference ID field will be associated with foreign table.

`Join::hasMany()`
: same as `Model::hasMany` but condition for related model will be based on foreign table field and `Reference::their_field` will be set to $foreign_table.'_id'.

`Join::containsOne()`
: same as `Model::containsOne` but the data will be stored in a field inside foreign table.

`Join::containsMany()`
: same as `Model::containsMany` but the data will be stored in a field inside foreign table.

Create and Delete behavior

Updating joined records are simple, but when it comes to creation and deletion, there are some conditions. First we look at dependency. If master table contains id of a foreign table, then foreign table record must be created first, so that we can store its ID in a master table. If the join is reversed, the master record is created first and then foreign record is inserted along with the value of master id.

When it comes to deleting record, there are three possible conditions:

1. [delete_behaivour = cascade, reverse = false] If we are using strong join and master table contains ID of foreign table then foreign master table record is deleted first. Foreign table record is deleted after. This is done to avoid error with foreign constraints.
2. [delete_behaviour = cascade, reverse = true] If we are using strong join and foreign table contains ID of master table, then foreign table record is deleted first followed by the master table record.
3. [delete_behaviour = ignore, reverse = false] If we are using weak join and the master table contains ID of foreign table then master table is deleted first. Foreign table record is not deleted.
4. [delete_behaviour = setnull, reverse = true] If we are using weak join and foreign table contains ID of master table, then foreign table is updated to set ID of master table to NULL first. Then the master table record is deleted.

Based on the way how you define join an appropriate strategy is selected and Join will automatically decide on $delete_behaviour and $reverse values. There are situations, however when it's impossible to determine in which order the operations have to be performed. A good example is when you define both master/foreign fields.

In this case system will default to "reverse=false" and will delete master record first, however you can specify a different value for "reverse".

Sometimes it's also sensible to set delete_behaviour = ignore and perform your own delete operation yourself.

Implementation Detail

Joins are implemented like this:

- all the fields that has 'join' property set will not be saved into default table by default driver
- join will add either *beforeInsert* or *afterInsert* hook inside your model. When save is executed, it will execute additional query to update foreign table.
- while $model->id stores the ID of the main table active record, $join->id stores ID of the foreign record and will be used when updating.
- option 'delete_behaviour' is 'cascade' for strong joins and 'ignore' for weak joins, but you can set some other value. If you use "setnull" value and you are using reverse join, then foreign table record will not be updated, but value of the foreign field will be set to null.

class `Join_SQL`

SQL-specific joins

When your model is associated with SQL-capable driver, then instead of using 'Join' class, the 'Join_SQL' is used instead. This class is designed to improve loading technique, because SQL vendors can query multiple tables simultaneously.

Vendors that cannot do JOINs will have to implement compatibility by pulling data from collections in a correct order.

Implementation Details

- although some SQL vendors allow update .. join .. syntax, this will not be used. That is done to ensure better compatibility.
- when field has the 'join' option set, trying to convert this field into expression will prefix the field properly with the foreign table alias.

- join will be added in all queries
- strong join can potentially reduce your data-set as it exclude table rows that cannot be matched with foreign table row.

Specifying complex ON logic

When you're dealing with SQL drivers, you can specify dsqlExpression for your "on" clause:

```
$stats = $user->join('stats', [
    'on'=>$user->expr('year({}) = _st.year'),
    'foreign_alias'=>'_st'
]);
```

You can also specify `'on'=>false` then the ON clause will not be used at all and you'll have to add additional where() condition yourself.

`foreign_alias` can be specified and will be used as table alias and prefix for all fields. It will default to `"_".$foreign_table[0]`. Agile Data will also resolve situations when multiple tables have same first character so the prefixes will be named '_c' ,'_c_2', '_c_3' etc.

Additional arguments accepted by SQL joins are:

- 'type' - will be "inner" for strong join and "left" for weak join, but you can specify other type of join.

CHAPTER 14

Hooks

class Model

Please never use `$this` inside your hook to refer to the model. The model is always passed as a first argument. If you ever use `$this` then your model will perform very weirdly when cloned:

```
$m->addHook('beforeSave', function($m) {
    $this['name'] = 'John';  // WRONG!
    $m['surname'] = 'Smith'; // GOOD
});

$m->insert([]);
// Will save into DB:  ['surname'=>'Smith'];

echo $m['name'];    // Will contain 'John', which it shouldn't
                    // because insert() is not supposed to affect active record
```

afterLoad hook

You can return false from afterLoad hook to prevent yielding of particular data rows.

Use it like this:

```
$model->addHook('afterLoad', function ($m) {
```

if ($m['date'] < $m->date_from) { $m->breakHook(false); // will not yield such data row

} // otherwise yields data row

});

Also this approach can be used to prevent data row to be loaded. If you return false from afterLoad hook, then record which we just loaded will be instantly unloaded. This can be helpful in some cases.

More on hooks

Coming soon

CHAPTER 15

Advanced Topics

Agile Data allow you to implement various tricks.

Audit Fields

If you wish to have a certain field inside your models that will be automatically changed when the record is being updated, this can be easily implemented in Agile Data.

I will be looking to create the following fields:

- created_dts
- updated_dts
- created_by_user_id
- updated_by_user_id

To implement the above, I'll create a new class:

```
class Controller_Audit {

    use \atk4\core\InitializerTrait {
        init as _init;
    }
    use \atk4\core\TrackableTrait;
    use \atk4\core\AppScopeTrait;

}
```

TrackableTrait means that I'll be able to add this object inside model with `$model->add(new Controller_Audit())` and that will automatically populate $owner, and $app values (due to AppScopeTrait) as well as execute init() method, which I want to define like this:

```
public function init() {
    $this->_init();

    if(isset($this->owner->no_audit)){
        return;
    }

    $this->owner->addField('created_dts', ['type'=>'datetime', 'default'=>date('Y-m-d␣
↪H:i:s')]);

    $this->owner->hasOne('created_by_user_id', 'User');
    if(isset($this->app->user) and $this->app->user->loaded()) {
        $this->owner->getElement('created_by_user_id')->default = $this->app->user->
↪id;
    }

    $this->owner->hasOne('updated_by_user_id', 'User');

    $this->owner->addField('updated_dts', ['type'=>'datetime']);

    $this->owner->addHook('beforeUpdate', function($m, $data) {
        if(isset($this->app->user) and $this->app->user->loaded()) {
            $data['updated_by'] = $this->app->user->id;
        }
        $data['updated_dts'] = date('Y-m-d H:i:s');
    });
}
```

In order to add your defined behavior to the model. The first check actually allows you to define models that will bypass audit altogether:

```
$u1 = new Model_User($db);    // Model_User::init() includes audit

$u2 = new Model_User($db, ['no_audit' => true]);  // will exclude audit features
```

Next we are going to define 'created_dts' field which will default to the current date and time.

The default value for our 'created_by_user_id' field would depend on a currently-logged in user, which would typically be accessible through your application. AppScope allows you to pass $app around through all the objects, which means that your Audit Controller will be able to get the current user.

Of course if the application is not defined, no default is set. This would be handy for unit tests where you could manually specify the value for this field.

The last 2 fields (update_*) will be updated through a hook - beforeUpdate() and will provide the values to be saved during `save()`. beforeUpdate() will not be called when new record is inserted, so those fields will be left as "null" after initial insert.

If you wish, you can modify the code and insert historical records into other table.

Soft Delete

Most of the data frameworks provide some way to enable 'soft-delete' for tables as a core feature. Design of Agile Data makes it possible to implement soft-delete through external controller. There may be a 3rd party controller for comprehensive soft-delete, but in this section I'll explain how you can easily build your own soft-delete controller for Agile Data (for educational purposes).

Start by creating a class:

```
class Controller_SoftDelete {

    use \atk4\core\InitializerTrait {
        init as _init;
    }
    use \atk4\core\TrackableTrait;

    function init() {
        $this->_init();

        if(isset($this->owner->no_soft_delete)){
            return;
        }

        $this->owner->addField('is_deleted', ['type'=>'boolean']);

        if (isset($this->owner->deleted_only)) {
            $this->owner->addCondition('is_deleted', true);
            $this->owner->addMethod('restore', $this);
        }else{
            $this->owner->addCondition('is_deleted', false);
            $this->owner->addMethod('softDelete', $this);
        }
    }

    function softDelete($m) {
        if (!$m->loaded()) {
            throw new \atk4\core\Exception(['Model must be loaded before soft-deleting
↪', 'model'=>$m]);
        }

        $id = $m->id;
        if ($m->hook('beforeSoftDelete') === false) {
            return $m;
        }

        $rs = $m->reload_after_save;
        $m->reload_after_save = false;
        $m->save(['is_deleted'=>true])->unload();
        $m->reload_after_save = $rs;

        $m->hook('afterSoftDelete', [$id]);
        return $m;
    }

    function restore($m) {
        if (!$m->loaded()) {
            throw new \atk4\core\Exception(['Model must be loaded before restoring',
↪'model'=>$m]);
        }

        $id = $m->id;
        if ($m->hook('beforeRestore') === false) {
            return $m;
        }

        $rs = $m->reload_after_save;
```

```
        $m->reload_after_save = false;
        $m->save(['is_deleted'=>false])->unload();
        $m->reload_after_save = $rs;

        $m->hook('afterRestore', [$id]);
        return $m;
    }
}
```

This implementation of soft-delete can be turned off by setting model's property 'deleted_only' to true (if you want to recover a record).

When active, a new field will be defined 'is_deleted' and a new dynamic method will be added into a model, allowing you to do this:

```
$m = new Model_Invoice($db);
$m->load(10);
$m->softDelete();
```

The method body is actually defined in our controller. Notice that we have defined 2 hooks - beforeSoftDelete and afterSoftDelete that work similarly to beforeDelete and afterDelete.

beforeSoftDelete will allow you to "break" it in certain cases to bypass the rest of method, again, this is to maintain consistency with the rest of before* hooks in Agile Data.

Hooks are called through the model, so your call-back will automatically receive first argument $m, and afterSoftDelete will pass second argument - $id of deleted record.

I am then setting reload_after_save value to false, because after I set 'is_deleted' to false, $m will no longer be able to load the record - it will fall outside of the DataSet. (We might implement a better method for saving records outside of DataSet in the future).

After softDelete active record is unloaded, mimicking behavior of delete().

It's also possible for you to easily look at deleted records and even restore them:

```
$m = new Model_Invoice($db, ['deleted_only'=>true]);
$m->load(10);
$m->restore();
```

Note that you can call $m->delete() still on any record to permanently delete it.

Soft Delete that overrides default delete()

In case you want $m->delete() to perform soft-delete for you - this can also be achieved through a pretty simple controller. In fact I'm reusing the one from before and just slightly modifying it:

```
class Controller_SoftDelete {

    use \atk4\core\InitializerTrait {
        init as _init;
    }
    use \atk4\core\TrackableTrait;

    function init() {
        $this->_init();

        if(isset($this->owner->no_soft_delete)){
```

```
            return;
        }

        $this->owner->addField('is_deleted', ['type'=>'boolean']);

        if (isset($this->owner->deleted_only)) {
            $this->owner->addCondition('is_deleted', true);
            $this->owner->addMethod('restore', $this);
        } else {
            $this->owner->addCondition('is_deleted', false);
            $this->owner->addHook('beforeDelete', [$this, 'softDelete'], null, 100);
        }
    }

    function softDelete($m) {
        if (!$m->loaded()) {
            throw new \atk4\core\Exception(['Model must be loaded before soft-deleting
↪', 'model'=>$m]);
        }

        $id = $m->id;

        $rs = $m->reload_after_save;
        $m->reload_after_save = false;
        $m->save(['is_deleted'=>true])->unload();
        $m->reload_after_save = $rs;

        $m->hook('afterDelete', [$id]);

        $m->breakHook(false); // this will cancel original delete()
    }

    function restore($m) {
        if (!$m->loaded()) {
            throw new \atk4\core\Exception(['Model must be loaded before restoring',
↪'model'=>$m]);
        }

        $id = $m->id;
        if ($m->hook('beforeRestore') === false) {
            return $m;
        }

        $rs = $m->reload_after_save;
        $m->reload_after_save = false;
        $m->save(['is_deleted'=>false])->unload();
        $m->reload_after_save = $rs;

        $m->hook('afterRestore', [$id]);
        return $m;
    }
}
```

Implementation of this controller is similar to the one above, however instead of creating softDelete() it overrides the delete() method through a hook. It will still call 'afterDelete' to mimic the behavior of regular delete() after the record is marked as deleted and unloaded.

You can still access the deleted records:

```
$m = new Model_Invoice($db, ['deleted_only'=>true]);
$m->load(10);
$m->restore();
```

Calling delete() on the model with 'deleted_only' property will delete it permanently.

Creating Unique Field

Database can has UNIQUE constraint, but this does work if you use DataSet. For instance, you may be only able to create one 'Category' with name 'Book', but what if there is a soft-deleted record with same name or record that belongs to another user?

With Agile Data you can create controller that will ensure that certain fields inside your model are unique:

```
class Controller_UniqueFields {
    use \atk4\core\InitializerTrait {
        init as _init;
    }
    use \atk4\core\TrackableTrait;

    protected $fields = null;

    function init() {
        $this->_init();

        // by default make 'name' unique
        if (!$this->fields) {
            $this->fields = [$this->owner->title_field];
        }

        $this->owner->addHook('beforeSave', $this);
    }

    function beforeSave($m)
    {
        foreach ($this->fields as $field) {
            if ($m->dirty[$field]) {
                $mm = clone $m;
                $mm->addCondition($mm->id_field != $this->id);
                $mm->tryLoadBy($field, $m[$field]);

                if ($mm->loaded()) {
                    throw new \atk4\core\Exception(['Duplicate record exists', 'field
↪'=>$field, 'value'=>$m[$field]]);
                }
            }
        }
    }
}
```

As expected - when you add a new model the new values are checked against existing records. You can also slightly modify the logic to make addCondition additive if you are verifying for the combination of matched fields.

Creating Many to Many relationship

Depending on the use-case many-to-many relationships can be implemented differently in Agile Data. I will be focusing on the practical approach. My system has "Invoice" and "Payment" document and I'd like to introduce "invoice_payment" that can link both entities together with fields ('invoice_id', 'payment_id', and 'amount_closed'). Here is what I need to do:

1. Create Intermediate Entity - InvoicePayment

Create new Model:

```
class Model_InvoicePayment extends \atk4\data\Model {
    public $table='invoice_payment';
    function init()
    {
        parent::init();
        $this->hasOne('invoice_id', 'Model_Invoice');
        $this->hasOne('payment_id', 'Model_Payment');
        $this->addField('amount_closed');
    }
}
```

2. Update Invoice and Payment model

Next we need to define reference. Inside Model_Invoice add:

```
$this->hasMany('InvoicePayment');

$this->hasMany('Payment', [function($m) {
    $p = new Model_Payment($m->persistence);
    $j = $p->join('invoice_payment.payment_id');
    $j->addField('amount_closed');
    $j->hasOne('invoice_id', 'Model_Invoice');
}, 'their_field'=>'invoice_id']);

$this->addHook('beforeDelete',function($m){
    $m->ref('InvoicePayment')->action('delete')->execute();

    // If you have important per-row hooks in InvoicePayment
    // $m->ref('InvoicePayment')->each('delete');
});
```

You'll have to do a similar change inside Payment model. The code for '$j->' have to be duplicated until we implement method Join->importModel().

3. How to use

Here are some use-cases. First lets add payment to existing invoice. Obviously we cannot close amount that is bigger than invoice's total:

```
$i->ref('Payment')->insert([
    'amount'=>$paid,
    'amount_closed'=> min($paid, $i['total']),
```

```
    'payment_code'=>'XYZ'
]);
```

Having some calculated fields for the invoice is handy. I'm adding *total_payments* that shows how much amount is closed and *amount_due*:

```
// define field to see closed amount on invoice
$this->hasMany('InvoicePayment')
    ->addField('total_payments', ['aggregate'=>'sum', 'field'=>'amount_closed']);
$this->addExpression('amount_due', '[total]-coalesce([total_payments],0)');
```

Note that I'm using coalesce because without InvoicePayments the aggregate sum will return NULL. Finally let's build allocation method, that allocates new payment towards a most suitable invoice:

```
// Add to Model_Payment
function autoAllocate()
{
    $client = $this->ref['client_id'];
    $invoices = $client->ref('Invoice');

    // we are only interested in unpaid invoices
    $invoices->addCondition('amount_due', '>', 0);

    // Prioritize older invoices
    $invoices->setOrder('date');

    while($this['amount_due'] > 0) {

        // See if any invoices match by 'reference';
        $invoices->tryLoadBy('reference', $this['reference']);

        if (!$invoices->loaded()) {

            // otherwise load any unpaid invoice
            $invoices->tryLoadAny();

            if(!$invoices->loaded()) {

                // couldn't load any invoice.
                return;
            }
        }

        // How much we can allocate to this invoice
        $alloc = min($this['amount_due'], $invoices['amount_due'])
        $this->ref('InvoicePayment')->insert(['amount_closed'=>$alloc, 'invoice_id'=>
↪$invoices->id]);

        // Reload ourselves to refresh amount_due
        $this->reload();
    }
}
```

The method here will prioritize oldest invoices unless it finds the one that has a matching reference. Additionally it will allocate your payment towards multiple invoices. Finally if invoice is partially paid it will only allocate what is due.

Creating Related Entity Lookup

Sometimes when you add a record inside your model you want to specify some related records not through ID but through other means. For instance, when adding invoice, I want to make it possible to specify 'Category' through the name, not only category_id. First, let me illustrate how can I do that with category_id:

```
class Model_Invoice extends \atk4\data\Model {
    function init() {

        parent::init();

        ...

        $this->hasOne('category_id', 'Model_Category');

        ...
    }
}

$m = new Model_Invoice($db);
$m->insert(['total'=>20, 'client_id'=>402, 'category_id'=>6]);
```

So in situations when client_id and category_id is not known (such as import or API call) this approach will require us to perform 2 extra queries:

```
$m = new Model_Invoice($db);
$m->insert([
    'total'=>20,
    'client_id'=>$m->ref('client_id')->loadBy('code', $client_code)->id,
    'category_id'=>$m->ref('category_id')->loadBy('name', $category)->id,
]);
```

The ideal way would be to create some "non-persistable" fields that can be used to make things easier:

```
$m = new Model_Invoice($db);
$m->insert([
    'total'=>20,
    'client_code'=>$client_code,
    'category'=>$category
]);
```

Here is how to add them. First you need to create fields:

```
$this->addField('client_code', ['never_persist'=>true]);
$this->addField('client_name', ['never_persist'=>true]);
$this->addField('category', ['never_persist'=>true]);
```

I have declared those fields with never_persist so they will never be used by persistence layer to load or save anything. Next I need a beforeSave handler:

```
$this->addHook('beforeSave', function($m) {
    if(isset($m['client_code']) && !isset($m['client_id'])) {
        $cl = $this->refModel('client_id');
        $cl->addCondition('code',$m['client_code']);
        $m['client_id'] = $cl->action('field',['id']);
    }
```

```
    if(isset($m['client_name']) && !isset($m['client_id'])) {
        $cl = $this->refModel('client_id');
        $cl->addCondition('name', 'like', $m['client_name']);
        $m['client_id'] = $cl->action('field',['id']);
    }

    if(isset($m['category']) && !isset($m['category_id'])) {
        $c = $this->refModel('category_id');
        $c->addCondition($c->title_field, 'like', $m['category']);
        $m['category_id'] = $c->action('field',['id']);
    }
});
```

Note that isset() here will be true for modified fields only and behaves differently from PHP's default behavior. See documentation for Model::isset

This technique allows you to hide the complexity of the lookups and also embed the necessary queries inside your "insert" query.

Fallback to default value

You might wonder, with the lookup like that, how the default values will work? What if the user-specified entry is not found? Lets look at the code:

```
if(isset($m['category']) && !isset($m['category_id'])) {
    $c = $this->refModel('category_id');
    $c->addCondition($c->title_field, 'like', $m['category']);
    $m['category_id'] = $c->action('field',['id']);
}
```

So if category with a name is not found, then sub-query will return "NULL". If you wish to use a different value instead, you can create an expression:

```
if(isset($m['category']) && !isset($m['category_id'])) {
    $c = $this->refModel('category_id');
    $c->addCondition($c->title_field, 'like', $m['category']);
    $m['category_id'] = $this->expr('coalesce([],[])',[
        $c->action('field',['id']),
        $m->getElement('category_id')->default
    ]);
}
```

The beautiful thing about this approach is that default can also be defined as a lookup query:

```
$this->hasOne('category_id','Model_Category');
$this->getElement('category_id')->default =
    $this->refModel('category_id')->addCondition('name','Other')
        ->action('field',['id']);
```

Inserting Hierarchical Data

In this example I'll be building API that allows me to insert multi-model information. Here is usage example:

```
$invoice->insert([
    'client'=>'Joe Smith',
    'payment'=>[
        'amount'=>15,
        'ref'=>'half upfront',
    ],
    'lines'=>[
        ['descr'=>'Book','qty'=>3, 'price'=>5]
        ['descr'=>'Pencil','qty'=>1, 'price'=>10]
        ['descr'=>'Eraser','qty'=>2, 'price'=>2.5]
    ],
]);
```

Not only 'insert' but 'set' and 'save' should be able to use those fields for 'payment' and 'lines', so we need to first define those as 'never_persist'. If you curious about client lookup by-name, I have explained it in the previous section. Add this into your Invoice Model:

```
$this->addField('payment',['never_persist'=>true]);
$this->addField('lines',['never_persist'=>true]);
```

Next both payment and lines need to be added after invoice is actually created, so:

```
$this->addHook('afterSave', function($m){
    if(isset($m['payment'])) {
        $m->ref('Payment')->insert($m['payment']);
    }

    if(isset($m['lines'])) {
        $m->ref('Line')->import($m['lines']);
    }
});
```

You should never call save() inside afterSave hook, but if you wish to do some further manipulation, you can reload a clone:

```
$mm = clone $m;
$mm->reload();
if ($mm['amount_due'] == 0) $mm->save(['status'=>'paid']);
```

Related Record Conditioning

Sometimes you wish to extend one Model into another but related field type can also change. For example let's say we have Model_Invoice that extends Model_Document and we also have Model_Client that extends Model_Contact.

In theory Document's 'contact_id' can be any Contact, however when you create 'Model_Invoice' you wish that 'contact_id' allow only Clients. First, lets define Model_Document:

```
$this->hasOne('client_id', 'Model_Contact');
```

One option here is to move 'Model_Contact' into model property, which will be different for the extended class:

```
$this->hasOne('client_id', $this->client_class);
```

Alternatively you can replace model in the init() method of Model_Invoice:

```
$this->getRef('client_id')->model = 'Model_Client';
```

You can also use array here if you wish to pass additional information into related model:

```
$this->getRef('client_id')->model = ['Model_Client', 'no_audit'=>true];
```

Combined with our "Audit" handler above, this should allow you to relate with deleted clients.

The final use case is when some value inside the existing model should be passed into the related model. Let's say we have 'Model_Invoice' and we want to add 'payment_invoice_id' that points to 'Model_Payment'. However we want this field only to offer payments made by the same client. Inside Model_Invoice add:

```
$this->hasOne('client_id', 'Client');

$this->hasOne('payment_invoice_id', function($m){
    return $m->ref('client_id')->ref('Payment');
});

/// how to use

$m = new Model_Invoice($db);
$m['client_id'] = 123;

$m['payment_invoice_id'] = $m->ref('payment_invoice_id')->tryLoadAny()->id;
```

In this case the payment_invoice_id will be set to ID of any payment by client 123. There also may be some better uses:

```
$cl->ref('Invoice')->each(function($m) {

    $m['payment_invoice_id'] = $m->ref('payment_invoice_id')->tryLoadAny()->id;
    $m->save();

});
```

Narrowing Down Existing References

Agile Data allow you to define multiple references between same entities, but sometimes that can be quite useful. Consider adding this inside your Model_Contact:

```
$this->hasMany('Invoice', 'Model_Invoice');
$this->hasMany('OverdueInvoice', function($m){
    return $m->ref('Invoice')->addCondition('due','<',date('Y-m-d'))
});
```

This way if you extend your class into 'Model_Client' and modify the 'Invoice' reference to use different model:

```
$this->getRef('Invoice')->model = 'Model_Invoice_Sale';
```

The 'OverdueInvoice' reference will be also properly adjusted.

CHAPTER 16

Loading and Saving CSV Files

class `Persistence_CSV`

Agile Data can operate with CSV files for data loading, or saving. The capabilities of Persistence_CSV are limited to the following actions:

- open any CSV file, use column mapping
- identify which column is corresponding for respective field
- support custom mapper, e.g. if date is stored in a weird way
- support for CSV files that have extra lines on top/bottom/etc
- listing/iterating
- adding a new record

Setting Up

When creating new persistence you must provide a valid URL for the file that might be stored either on a local system or use a remote file scheme (ftp://...). The file will not be actually opened unless you perform load/save operation:

```
$p = new Persistence_CSV('myfile.csv');

$u = new Model_User($p);
$u->tryLoadAny();   // actually opens file and finds first record
```

Exporting and Importing data from CSV

You can take a model that is loaded from other persistence and save it into CSV like this. The next example demonstrates a basic functionality of SQL database export to CSV file:

```
$db = new Persistence_SQL($pdo);
$csv = new Persistence_CSV('dump.csv');

$m = new Model_User($db);

foreach (new Model_User($db) as $m) {
    $m->withPersistence($csv)->save();
}
```

Theoretically you can do few things to tweak this process. You can specify which fields you would like to see in the CSV:

```
foreach (new Model_User($db) as $m) {
    $m->withPersistence($csv)
        ->onlyFields(['id','name','password'])
        ->save();
}
```

Additionally if you want to use a different column titles, you can:

```
foreach (new Model_User($db) as $m) {
    $m_csv = $m->withPersistence($csv);
    $m_csv->onlyFields(['id', 'name', 'password'])
    $m_csv->getElement('name')->actual = 'First Name';
    $m_csv->save();
}
```

Like with any other persistence you can use typecasting if you want data to be stored in any particular format.

The examples above also create object on each iteration, that may appear as a performance inefficiency. This can be solved by re-using CSV model through iterations:

```
$m = new Model_User($db);
$m_csv = $m->withPersistence($csv);
$m_csv->onlyFields(['id', 'name', 'password'])
$m_csv->getElement('name')->actual = 'First Name';

foreach ($m as $m_csv) {
    $m_csv->save();
}
```

This code can be further simplified if you use import() method:

```
$m = new Model_User($db);
$m_csv = $m->withPersistence($csv);
$m_csv->onlyFields(['id', 'name', 'password'])
$m_csv->getElement('name')->actual = 'First Name';
$m_csv->import($m);
```

Naturally you can also move data in the other direction:

```
$m = new Model_User($db);
$m_csv = $m->withPersistence($csv);
$m_csv->onlyFields(['id', 'name', 'password'])
$m_csv->getElement('name')->actual = 'First Name';

$m->import($m_csv);
```

Only the last line changes and the data will now flow in the other direction.

CHAPTER 17

Indices and tables

- genindex
- search

Index

O

P

R

S

T

U

W