

Introduction to Agile Toolkit

The Agile Toolkit is a software development platform for creating Web Software using PHP language. Unlike many other PHP frameworks today, Agile Toolkit focuses on depth, simplification and sophistication. The goals of Agile Toolkit is make web development more efficient without introducing overheads typically associated with frameworks.

Agile Toolkit is based on some philosophical concepts for easier learning. By Learning the way of Agile Toolkit, you will discover a new approach to build web applications in general. Many concepts might be new or might seem unusual, but you will find the dynamic synergy behind them and the rest of the toolkit.

In this chapter I will explain what motivated me to create Agile Toolkit and why other frameworks failed to provide a good-enough solution. I will also look at the number of challenges developers face daily and how Agile Toolkit solves them.

The biggest mistake you could make when learning Agile Toolkit is if you assume it to be yet another framework. There are fundamental differences which I will attempt to communicate to you in this chapter.

The rest of the book will focus on all the components of Agile Toolkit and how the low-level implementation is abstracted and used to build a higher level of the toolkit. You will be able to see Agile Toolkit in three dimensions.

Based on version 4.2.3 of Agile Toolkit # The history of “broken” PHP

Not many remember what web 1.0 applications were like. In early '90ties most of web sites on the Internet were static and not interactive. As demand increased Common Gateway Interface (CGI) and soon Perl were becoming incredibly popular.

Two major problems were getting in the way of web software: slow server speed and poor support for JavaScript in the browsers.

PHP3 was released as a promising language and quickly gained popularity. Even today PHP powers the majority of World Wide Web. Yet many consider PHP a poor choice for software development.

The initial versions of PHP language were mimicking C and Perl. A lot of emphasis was on the execution speed and diversity of built-in functionality in PHP. Objects were becoming more and more popular, but they always were optional, just like C++ applications. What made PHP the most popular language and how to make it more attractive for professional developers? These were my goals in creating Agile Toolkit.

One of the main concepts of PHP is it's ability to combine HTML and code. While it is appreciated by the novice developers, this feature is at fault for the poor practices in HTML and PHP separation in application.

Many developers today consider PHP a poor or “broken” language.

Fixing PHP with Agile Toolkit Philosophy

Today there are too many ways to develop a web applications in PHP. What if we just agree on a single “best” way to do things, which would be sufficiently flexible and would fit 90% of the use-cases? That's what Agile Toolkit does philosophically. In the very core it defines the set of principles we use for application development. Agile Toolkit PHP code follows the principles and so will be your code.

If you are a manager of a project and you choose to use Agile Toolkit in your environment adopting the psychology, it will be much easier for new recruits to learn the toolkit. Your code will be consistent no matter how big your project is and if a developer with understanding of Agile Toolkit will be able to look inside any part of your website and instantly understand how it works.

What's more important, is that your developers can also look inside the framework itself, add-ons or any other software built on Agile Toolkit and understand it.

I myself participate as IT manager or consultant in several companies over a decade and this rule have helped us keep all our project in a very maintainable state.

So when PHP comes up with a set of features, we would carefully review them and decide if we want or don't want to use them in the toolkit and globally. Here are some of the features Agile Toolkit advises against using:

- Mixing up PHP and HTML with `<?php/?>` tags. Always use template engine.
- Static classes and constants. Those are not portable and not fully object-oriented. Use of singletons is also discouraged.
- Over-use of namespaces. We use name-spaces very mildly to separate the core code from add-ons. Avoid "use" keyword as it makes your source codes very confusing.
- Explicit includes. There is only one file you include with Agile Toolkit. The rest is being included automatically by converting class name into filename.
- Private properties and final classes. Unless in rare cases, you can't predict how your class will be used. Embrace extensibility and use "public \$_prop" if you need to highlight that some property is public.
- Keep the size of your classes reasonable. Don't make classes longer than a thousands lines of code.
- Don't copy-paste your code. Remember to refactor it.

Inspiration for Agile Toolkit

Agile Toolkit draws it's inspiration from a several non-PHP sources.

jQuery

While many PHP has similarities to the JavaScript and Java libraries, the closest resembling framework for Agile Toolkit is jQuery. Advanced use of objects, short and concise syntax, heavy method overloading, philosophy, intuitive approach and even similar syntax brings both libraries close together.

```
// jQuery Code
$('li').hide().fadeIn();

// Agile Toolkit - DSQL
$name = $dsql->table('user')->field('name')->getOne();

// Agile Toolkit jQuery Chain
$form->getElement('name')->js('click')->val('');

// Agile Toolkit Model
$user->load(1)->set('name','John')->save();
```

Desktop Toolkits

Agile Toolkit has a very strong User Interface framework. It based on the principles used by desktop object-oriented toolkits:

```
// QT
fileMenu = menuBar()->addMenu(tr("&File"));
fileMenu->addAction(newAct);
```

```
// Agile Toolkit
$menu = $page->add('Menu');
$menu->addMenuItem('Users');
```

Agile Toolkit also allows it's developers to easily breach gap between the browser and the server by automatically using AJAX for call-backs.

```
// Agile Toolkit
$button = $page->add('Button')->set('Delete');
if($button->isClicked()){
    $model->delete();
    $this->js()->univ()->alert('Item Deleted')->execute();
}
```

Model-View-Presenter

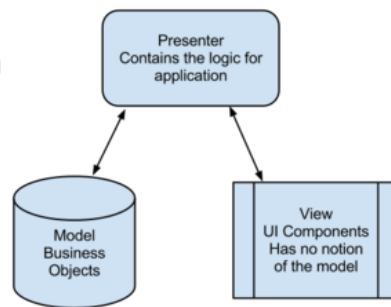
Model-View-Presenter is a derivative of the MVC software pattern. The MVP is

mostly used for building User Interfaces.

A good example in Agile Toolkit is the way how you use a form. You would need to have a three components:

Page is the type of object which receives control from the Application in response to the requested URL and is responsible for a single page of your web application.

Form is a class for displaying a generic form on a page. Finally a Model is a class representing a business object, such as "user".



```
$page->add('Form')->setModel('User');
```

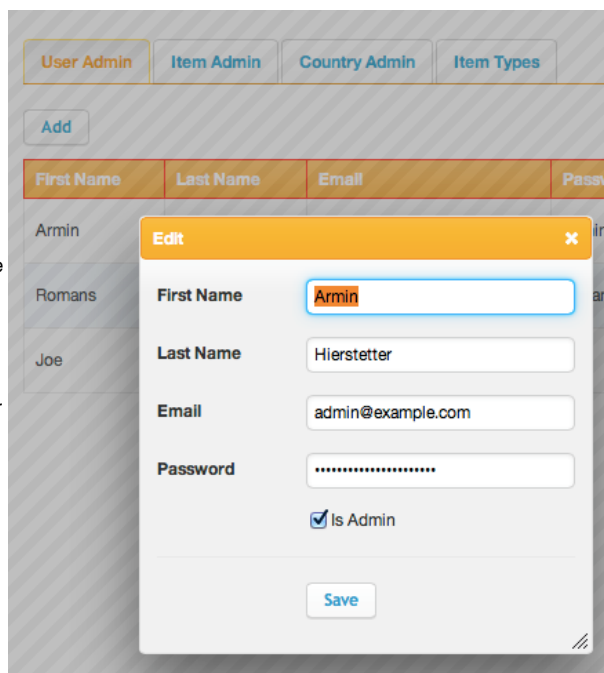
Window Systems

A typical desktop windowing systems consists of views.

By definition a view is a region on the screen which can display itself. Often view would contain other views which it relies on when it's rendering time. A good example is a dialog box, which contains buttons.

Agile Toolkit relies on a similar concept in the way how it produces HTML and JavaScript. The HTML for the application is recursively rendered by the page and all the views you would add to the page.

The screenshot here demonstrates a page containing Tabs which contains CRUD which contains Grid. When Edit button is clicked then a new window displays another Page containing Form which contains Fields and Button. All of the above is easily done by this code:



```
$tabs = $this->add('Tabs');
$tab = $tabs->addTab('User Admin');

$user_crud=$tab->add('CRUD');
$user_crud->setModel('User');
```

Confused Object-Orientation

I often hear from developers how good they are with Object Oriented design and then they design all their classes without extending anything. Similarly - many frameworks claim to be object-oriented, yet they fail to grasp the core concepts of Object-Oriented design:

Abstraction.

When you write object-oriented code, you don't just do it for the sake of writing. You do it to hide the complexity of your implementation and make it interchangeable.

For example, in Agile Toolkit you can make take a sophisticated web page and replace "Grid" view with "Lister" view. It will retain most of it's functionality but will be presented differently. Similarly you can substitute one model with another and have the UI automatically adopt to it.

The other big problem is that there is a large portion of libraries for PHP which do not rely on other libraries. They produce huge monolithic classes which can offer very little. A good example would be all those API libraries you download from facebook, twitter, etc. They pack up requests into CURL and will send them away within a 1000-line class, while the same functionality can be done using native SOAP implementation in 10 lines. I'm not saying SOAP is good, but it's a good example how you can rely on existing technology to make your life simpler.

With Agile Toolkit you will have a huge amount of objects which you'll rely on. Clearly their implementation is not ideal, they might lack some features but any class is interchangeable and can be substituted by your own implementation.

This modularity is a huge benefit of Agile Toolkit and once you learn how to override existing methods in classes, you will have no boundaries in Agile Toolkit.

Legacy and Backwards Compatibility

As the frameworks make it to their 2.0, many of them loose most of their adopters who simply don't want to learn completely new framework. This will not happen with Agile Toolkit and here is why:

1. Agile Toolkit is already in it's 4.X versions. There have been a lot of movement in the past and a lot of things have settled down.
2. Most of things in Agile Toolkit is interchangeable. If you like the implementation of "Menu" from a previous version of Agile Toolkit, it can still work.
3. New version come with compatibility controller, which can emulate functionality of a previous version.

Learning Agile Toolkit is a good investment as the framework will be there for a while and clearly have it's goals right.

Focusing on HOW versus WHAT

Some features such as "timestamps" or "soft delete" are often used on web projects and many frameworks have added support for them. Agile Toolkit have introduced the concept of Conditions and Behaviors in Models and it allows a seamless implementation of timestamps and soft deletes.

Often Agile Toolkit would provide you a way how you can implement things on your own instead of coming up with all the possible solutions for your future problems.

Educating Developers

With many frameworks, when you ask "How can I implement X" the solution would be already built-in into a module. How can you implement CRUD in Yii? Obviously - use the generator. What if you don't want to?

In Agile Toolkit we try to educated developers how each individual component is built and how he could create an alternative implementation. In a way - the core of Agile Toolkit is more open and more fluid.

Avoiding Over-Use Objects

If you have programmed in Smalltalk or Java you would know that all the primitive types (numbers, strings, chars) are objects. That's the decision in language design and the frameworks built in those languages extensively use objects for every possible thing.

In PHP it relies on a supplemental use of Objects. Strings, Numbers, Arrays are still a proper and awesome features of the language even through you can't extend them. You can't overload operations and that's fine. We understand that PHP is the way it is and we continue and Agile Toolkit does not attempt to artificially introduce a layer to wrap everything into a meaningless layer of objects.

As you write your logic in Agile Toolkit, you do things in the regular way. If you need to validate an email, you can use `var_filter` or a regexp. Agile Toolkit will not wrap it inside a validator object for you. If you want - you can create class like that yourself and distribute it as an add-on.

Agile Toolkit is What You Make of It

With Agile Toolkit you can develop command-line software. The features of Agile Toolkit is implemented by it's Application class and you can pick the one you like best. If don't need routing and database connection, use less-powerful application class:

```
include '../atk4/loader.php';
$api=new ApiWeb();
$api->add('jUI');
$f=$api->add('Form');
$f->addField('line','a')->set(2);
$f->addField('line','b')->set(2);
$f->addField('line','result')->set(2);
$f->addButton('sum');
if($f->isSubmitted()){
    $f->getElement('result')->js()
        ->val($f->get('a')+$f->get('b'))->execute();
}
```

Depth of Functionality

No doubt that many of you would look at the simple functionality of Agile Toolkit and will assume it's a basic framework. In reality Agile Toolkit uses a lot of sophisticated logic to make it tick in the best possible way.

Let's draw some lines between Laravel's Fluent Query Builder and Agile Toolkit's DSQL. On the surface they appear to be quite similar:

```
// Laravel
$users = DB::table('users')
    ->where('email', '=', 'example@gmail.com')
    ->where_in('id', array(1, 2, 3))
    ->get();

// DSQL
$this->api->db->dsql()->table('users')
    ->where('email', '=', 'example@gmail.com')
    ->where('id','in',array(1, 2, 3))
    ->get();
```

However both query builders are on completely different levels. DSQL is template-based recursive on-demand renderer while Fluent builds query as-you-define it. Here are some of the things DSQL does and Fluent does not: Support for subqueries; Recursive rendering; Custom class design; Modifying existing query; Native integration with Model Fields.

Because of design decisions in DSQL, it's possible to have a very fluid and dynamic Model / ORM implementation on top of it. Having said that, the implementations of both query builders is quite similar. # Challenges that Agile Toolkit solves

Agile Toolkit framework is not there only for a show. It is there to solve a real-life challenges a typical web developer is facing.

Secure by Design

How do you force your developer to produce secure software? One way would be to send him to several security conferences and lectures or even hire a hacker to review the code for a possible faults.

Agile Toolkit introduces the concept of "Secure by Design". It is a technique designed to create a simpler way which is also a simple way.

Majority of security problems arise from the lack of encoding as the content goes from one system to another. SQL databases controlled through a language constructed by string concatenation often is a source of many security issues.

The similar problem does not appear in noSQL databases which have object-oriented interaction techniques. Agile Toolkit introduces a object-oriented technique for interacting with SQL, JavaScript and HTML which is secure by default.

```
$ops = $_GET['arg'];
$this->add('H1')->set($ops);
$this->add('Button')->js('click')->univ()->alert($ops);
$this->api->dsq1()
    ->table('user')->where('name',$ops)->delete();
```

It might seem that the above code contain several vulnerabilities - JavaScript injection and SQL injection. However it's perfectly safe in Agile Toolkit. While it is advisable to validate the input data in Agile Toolkit, the way how developer interacts with HTML, JavaScript or SQL contains a proper escaping and pretty much eliminates any possible vulnerabilities without developer even being aware of them.

Secure Model Design

Models describe business entities in the database. For most of us RDBMS is a database of choice and while NoSQL solutions can be helpful sometimes, the SQL is used in majority of software projects.

To simplify interaction with the database there are two abstraction mechanisms in Agile Toolkit: Active Record and Dynamic SQL. Their combination brings the implementation of "Relational Models".

If you are familiar with "Views" in several SQL implementation, this is quite similar to models in Agile Toolkit. Model can select either all records from a SQL table or a sub-set limited by certain conditions. It can also interact with multiple tables simultaneously through joins or expressions.

The aspect of security is added by a sticky conditions. Once condition is applied to the model, the condition can no longer be removed. For example, if you set the following condition to your model

```
$model->addCondition('deleted',false);
```

then all the further interaction with the model will only affect records which are not flagged as deleted:

```
$model->load(5);    // will not delete deleted entry

$model->set('deleted',true)->save(); // will not save

foreach($model as $row){}    // will skip deleted entries

$cnt=$model->count()->getOne(); // will not count deleted

$model->dsq1()->set('name',123)
    ->update(); // will not update deleted entries
```

Scalability by Design

Many software platforms offer you a unique, interesting and even magical new ways of building web software, but as your logic becomes more and more complex, the performance start to crumble. Other frameworks could not carry less about performance and scalability of your software leaving it all in the hands of developer.

Agile Toolkit offers developer a ways how to keep their software incredibly scalable even as the data model becomes more and more complex. Consider this example:

```
// Count states from user addresses
$states=array();
foreach($user_model as $r){
    $address_model=$user_model->ref('address_id');
    $states[$address_model['state']]++;
}
```

This syntax is common in many Active-Record implementation. Traversing from the "user" model to the relevant "address" model executes one more query and the whole code generates N+1 queries and is highly ineffective.

Few of Active-Record implementation allow you to implement it by creating a separate model. Other implementation require you

to come up with a query yourself. Agile Toolkit allows you to tweak the model just for a specific query:

```
// Count states from user addresses
$states=array();
$user_model->join('address')->addField('state');
foreach($user_model as $r){
    $states[$user_model['state']]++;
}
```

The new approach will perform only a single query. There are many other ways to achieve the above in the single query with Agile Toolkit.

Reusable by Design

CRUD is a component present in many frameworks. Short for Create, Read, Update and Delete this component allows building a simple administrative interface for managing model data.

More often than not CRUD is a generator - a new excuse for avoiding "Don't Repeat Yourself" principle. Once you generate pages, you would then need to edit those pages yourself if you wish to change anything.

In Agile Toolkit CRUD is a View which heavily rely on Form and Grid views. While CRUD does offer some of the functionality, as a developer you can still access the Grid / Form components and modify them to your liking.

In Agile Toolkit there are no Code Generation!

So how do you add a new button / action to a standard CRUD in Agile Toolkit?

```
$crud=$page->add('CRUD');
$crud->setModel('User');
if($crud->grid){
    $crud->grid->addColumn('button','test');
    if($_GET['test']){
        $this->js()->univ()->alert('Testing: '.
            $crud->model
                ->load($_GET['test'])
                ->get('name')
            );
    }
}
```

In fact, many other complex controls and admins are in fact views which you can extend, redefine and interact with their elements directly. As you will develop your projects, you will also create re-usable components on a daily basis.

With just a few simple modifications any component in your code can be converted into add-on and published for others to use.

Collaborative by Design

Have you ever worked on as software project within a team? Have you ever found a code added by other developer in your files which was breaking some of your functionality? This happens often, but not in Agile Toolkit.

When you develop project in Agile Toolkit, you clearly separate things. For example you might have a dashboard page containing all sorts of views on them: recent purchases, account balance, current offer and advertisement block. Your dashboard page would most likely look like this:

Use of Agile Toolkit components in other Frameworks

Nowadays frameworks consist of independent components, which can be easily used outside of the framework. The same applies to Agile Toolkit, however the components are not as independent as elsewhere.

Agile Toolkit consists of large number of classes which can be used anywhere with the condition that you do have an application class.

The Core Framework

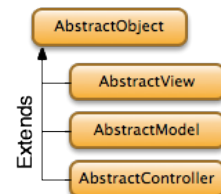
AbstractObject is a core class used in Agile Toolkit. This class is used by all the other classes in Agile Toolkit (except Exception classes) and offers number of core functionalities you can rely on. Further in this chapter I will include a code examples, where `$this` always refers to the object of Agile Toolkit.

Descendants of AbstractObject

There are 3 direct descendants of AbstractObject which are:

- AbstractView - parent of all the Views - object capable of rendering themselves into HTML
- AbstractModel - parent of all classes which represent individual entity.
- AbstractController - for all the classes which implement a library-like functionality.

This chapter will focus on those four core classes.



AbstractObject

add() method

While you might have used “new” operator before to create class instances, in Agile Toolkit you will use the operation called Adding.

```
$new_object = $owner->add('MyClass');
```

While the syntax is somewhat similar to the “new” operator, the fundamental difference is that the `$new_object` is created within the scope of it's `$owner`.

Depending on the class of the parent and child objects, there might be different effects. If you add Button into Page then sub-sequential rendering of a Page will also render a Button.

Newly created objects are also well aware of their owner-object:

```
$new_object -> owner === $owner; // knows who added it
```

The top-most Object

You might be curious. If you need an object to add another object, where is the top-most object coming from? The top-most object is called “Application” and it is initialized first.

Below is a sample command-line application in Agile Toolkit:

```
include 'atk4/loader.php';
$api = new ApiCLI();      // application object
$my_object = $api->add('MyClass');
$my_object->api === $api; // for any object
```

All objects can access the top-most Application object through “api” property.

Object's name

As you add more objects, each of them generates unique name. This name can be accessed through a “name” property. It is based on a parent object's name. A custom name can be specified as a second argument to `add()`, but if omitted, then the class name will be used instead.


```
$new_object = $owner->add('MyClass','foo');

echo $new_object->name;           // name_of_owner_foo
echo $new_object->short_name;    // foo
```

Unique names are important because that's how objects can identify themselves. This is often used in AJAX requests, HTML ID properties and selective rendering of HTML.

Note: you may change name of an object by calling `rename()`, but it must be used with caution as it may have some side-effects on a complex objects.

Element Tracking

When adding an object, the owner will create a note inside its “elements” property about the new object. This process is called element tracking. When you are adding one view into another, this connection will help to perform recursive rendering without you explicitly calling render of child objects. However the default behavior for non-view objects is not to keep track of elements.

The “elements” property of non-views would contain `'short_name'=>true`; association. Therefore if you lose the reference to your object (which is returned by `add()`) the object would be destroyed.

This helps Agile Toolkit to get rid of models, controllers when you no longer need them through the use of PHP garbage controller. There are however some exceptions, where tracking is necessary, such as Model Fields which needs to be associated with the Model object.

Method init()

When you add an object, Agile Toolkit will automatically execute method `init()` of a new object. This method is called after “owner”, “api” and “name” properties are set.

Below is the example from Strength Checker indicator:

```
class StrengthChecker extends View {
    function init(){
        parent::init();
        if(!$this->owner instanceof Form_Field_Password)
            throw $this
                ->exception('Must be added into Password field');
        // ....
    }
}
```

You will find that `init()` method can be redefined in ANY objects in Agile Toolkit. The fundamental principle of writing your own `init()` method is: **Avoid heavy-lifting. Do that in `render()` method instead, if possible.**

Addition of objects in Agile Toolkit is referred to by Object-Oriented developers as “Incapsulating”. Often functionality of \$this object is extended by a new object.

Return Value of add()

A call to `add()` will always return a new object of specified class. You can rely on that and chain additional calls to a returned value.

```
$page->add('Button')->setLabel('Click Me');
```

In a true jQuery spirit, most of the object methods will return reference to themselves (`$this`) so that you can do more chaining.

Indirect Adding of Objects

For convenience, objects may define new methods for adding certain types of objects. For example, Form has method called "addField" and grid has a method addButton();

```
$form->addButton('Click Me');  
// almost same as  
$form->add('Button')->setLabel('Click Me');
```

In reality those methods will do bit more than just calling add() for you.

Dependency Injection / Properties

If you specify second argument to add() an array, it will be used to initialize properties of a new object before init() is called. CRUD::init() adds Grid, but if you want to make it use a different class you can do it by specifying second argument.

```
$page->add('CRUD', array('grid_class' => 'MyGrid'))  
->setModel('User');
```

You can still specify name through array with the key of 'name'

Using namespaces / add-ons

Agile Toolkit allows you to add objects from add-ons. Each add-on resides in it's own namespace, so to add object you should use slash to separate namespace and class name:

```
$page->add('myaddon/MyClass');
```

Note: as you move code from your core app to add-on the format of using add() from within a namespace remains the same.

Working with Existing Objects

AbstractObject provides you with several handy methods to access objects you added into it.

```
$view = $page->add('View', 'myview');  
  
$page->hasElement('myview'); // returns $view or false  
  
$page->getElement('myview'); // returns $view or exception  
  
$view->destroy(); // removes object from parent and memory
```

Cloning and newInstance()

Models and Controllers of Agile Toolkit are generally OK to clone, however there is an alternative way: newInstance(). This method will add a new object of a same class added into the same parent.

If you experience problems with clone, try newInstance().

Adding Views

When you add a View into another View, you can specify two extra arguments for add() method. Third argument define where exactly HTML will appear and the fourth argument can override the template used by a View. I will further look into this when I'll be reviewing the chapter dedicated to Views.

String presentation

You can convert objects to string:

```
echo $page->add('View');  
// outputs Object View(name_of_view)
```

Adding object

If object is already created, you can use it as a first argument of an add() method. This will re-assign the object's owner property. Please use this functionality with caution.

```
$form = $page->add('Form');  
$field = $form->addField('line','foo');  
$frame = $form->add('View_Frame');  
$frame->add($field); // moves field inside frame
```

Model property

For any object you can call setModel() method. It will initialize the model if necessary and will assign it to the "model" property. Although it's easier to simply access the model property, you can also use getModel() method. You can specify object to setModel();

```
$grid = $page->add('Grid');  
$form = $page->add('Form');  
  
$grid->setModel('User'); // uses class Model_User  
$form->setModel($grid->model); // uses same object
```

Many objects redefine setModel() to perform additional binding of the model. When calling setModel() on a View, you can specify second argument, which is list of actual fields.

Using setController

You can use setController as a shortcut for add('Controller_..'). Again, you may use either a controller by name or an object. The name will be prefixed with 'Controller_'.

Session management

All objects in Agile Toolkit come with four methods to access session data: memorize(), learn(), forget() and recall(). You can specify the "key", which will be used in scope of a current object:

```
$obj1 = $this->add('MyObject');  
$obj2 = $this->add('MyObject');  
  
$obj1->memorize('test','foo');  
  
$obj2->recall('test'); //returns null  
$obj1->recall('test'); // returns foo
```

You can learn more about Agile Toolkit management under the Application section.

| Method | Arguments | Description |
|----------|------------------------------|---|
| memorize | name, value | Store value in session under name |
| recall | name, default | If value for name was stored previously return it, otherwise return default |
| forget | name | Remove previously memorized value from session |
| learn | name, value1, value2, value3 | Memorize first non-null argument. |

Exceptions

Agile Toolkit takes full advantage of exceptions. You can read more about exceptions in ...

To raise exception please use the exception() method of an object:

```
$obj = $this->add('MyObject');
throw $obj->exception('Something is Wrong');

// alternatively, add more info to exception
throw $obj->exception('Something is Wrong')
    ->addMoreInfo('foo',123);
```

My default objects through exception of a class "BaseException". You may use your own class:

```
class Exception_MyTest extends BaseException {
    function init(){
        parent::init();
        $this->addMoreInfo('thrown by',$this->owner->name);
    }
}

throw $obj->exception('Something is Wrong','MyTest');
```

Default exception type can also be specified on per-object basis through a property:

```
class Model_Test extends AbstractModel {
    public $default_exception='Exception_MyTest';
}

// or

$this->add('MyObject',array('default_exception'=>
    'Exception_MyTest'));
```

Third argument to exception() can specify a default exception code which can further be accessed through "code" property.

Hooks

Hooks is a callback implementation in Agile Toolkit. Hooks are defined throughout core classes and other controllers can use them to inject code.

For example:

```
// Add gettext() support into the app
$this->api->addHook('localizeString',function($obj,$str){
    return _($str);
});
```

The localizeString hook is called by many different objects and through adding a hook you can intercept the calls.

```
$obj = $this->add('MyClass');
$obj->addHook('foo',function($o){ return 1; });
$obj->addHook('foo',function($o){ return 2; });
$res = $obj->hook('foo'); // array(1, 2);
```

This example demonstrates how multiple hooks are called and how they return values. You can use method breakHook to override return value.

```
$obj = $this->add('MyClass');
$obj->addHook('foo',function($o){ return 1; });
$obj->addHook('foo',function($o){ $o->breakHook('bar'); });
$res = $obj->hook('foo'); // 'bar';
```

You should have noticed that all the hook receive reference to \$obj as a first argument. You can specify more arguments either through hook() or addHook()

```
$obj->addHook('foo',function($o,$a,$b,$c){
    return array($a,$b,$c);
}, array(3));
$res = $obj->hook('foo',array(1,2)); // array(array(1,2,3));
```

When calling addHook() the fourth argument is a priority. Default priority is 5, but by setting it lower you can have your hook called faster.

```
$obj = $this->add('MyClass');
$obj->addHook('foo',function($o){ return 1; });
$obj->addHook('foo',function($o){ return 2; },3);
$res = $obj->hook('foo'); // array(2, 1);
```

Note: in this example, the "3" was passed as 3rd argument not fourth. addHook automatically recognize non-array value as a priority if array with arguments is omitted. Argument omission is often used in Agile Toolkit methods.

When you are building object and you wish to make it extensible, adding a few hooks is always a good thing to do. You can also check the presence of any hooks and turn off default functionality:

```
function accountBlocked(){
    if(!$this->hook('accountBlocked'))
        $this->email('Your account have been blocked');
}
```

Without any hooks, hook() will return empty array.

Finally you can call removeHook to remove all hooks form a spot.

```
$obj = $this->add('MyClass');
$obj->addHook('foo',function($o){ return 1; });
$obj->removeHook('foo');
$res = $obj->hook('foo'); // array();
```

Note: If your object implements handlers for a few hooks and sets them inside init(), then after cloning such an object, it will not have the handlers cloned along with the object. Use of newInstance() should work fine.

Method Calling

PHP has a great way of extending object methods through a catch-all. Agile Toolkit implements catch-all for all of its objects which is then wrapped through some hooks to bring the functionality of addMethod(). It allows you to register method in any object dynamically. Note, that if such method already exists, you wouldn't be able to register a method. This is typically used for compatibility, when some methods are obsoleted and controller re-adds those methods back.

```
$obj = $this->add('MyClass');
$obj->addMethod('sum',function($o,$a,$b){ return $a+$b; });
```

Agile Toolkit also adds hasMethod() into all object as a preferred way to check if method exists inside object;

```
if($obj->hasMethod('sum')){
    $res = $obj->sum(2,3); // 5
}
```

You can also remove dynamically added methods with removeMethod();

Creating your first Command-Line application

You can create software for a variety of different targets with Agile Toolkit including command-line software or software which does not produce HTML.

Let's look at how to implement the simplest hello world in Agile Toolkit.

```
include 'atk4/loader.php';
$api = new ApiCLI();
echo "Hello World\n";
```

The creation of the API class does almost nothing and is a very lightweight operation. Once you have created your \$api class you can perform various things with it. For example, we can attempt to read a configuration parameter

```
include 'atk4/loader.php';
$api = new ApiCLI();
echo $api->getConfig('greeting',"Hello World\n");
```

This will open the config.php file in the same folder, and look for the following line:

```
$config['greeting']='Hello from Agile Toolkit';
```

and the output now would be different. Let's also connect to a MySQL database by adding the following line inside config.php:

```
$config['dsn']='mysql://root:secret@127.0.0.1/myproject';
// add to main PHP file:
$api->dbConnect();
```

I will revisit work with database in Chapter 3 - Data Model.# Template Engine

Web software commonly relies on template engines to separate HTML from the rest of the logic code. Agile Toolkit also contains a template engine which is used extensively. The engine is designed to be simple and quick. It resembles a DOM tree interaction, but is faster and simpler. Here is a template example:

```
<?greeting?>Hello, <?$name?>!! <?/greeting?>
```

Features of SMLite Template Engine

1. A passive template engine - template language contains no logic or complex code.
2. Tags define a placeholder where value can be inserted.
3. Regions represent the section inside a template, which may contain other tags or regions.

Tag is actually a self-closing Region, therefore SMLite does not distinguish between Tags and Regions. However for the sake of simpler understanding, I'll be calling them separately.

Basic Operations of SMLite

SMLite is an object, instance of SMLite class which can load the template from a file. Typically files have .html extension in Agile Toolkit and are located under "template" subdirectory.

Template can also be cloned from another template, in which case you can specify a region. Everything within that region will become a new template and any operations with new template will not affect the main template.

Finally - you can specify both the filename and the region. This allows you to define multiple templates in a single file.

Real-World workflow considerations

Concepts behind SMLite have formed over the course of 10 years by observing how designers work with templates and typical problems present in a conventional template systems.

- Extension .html makes it possible to open templates in browser or HTML editors directly (such as Dreamweaver) and edit them without risking to loose any data.
- The tags <?, ?> are commonly recognized by template parsers are unlikely to be a valid part of HTML document.
- Ability to initialize multiple SMLite copies from the same template file allows developer to preserve files in the way they when designer have created HTML file. This will allow designer to later do more modifications inside templates without wondering where did his code go.
- Lack of logic inside template files makes it simple for designers to read and understand tags. Often designers can create tags themselves without knowing application structure and relieve developer from this task.
- Ability to use tags as regions, gives ability to preserve original "Lorem Ipsum" text which was initially placed by designer, and then dynamically substitute it through the SMLite template.

Templates in Agile Toolkit are optimized for dynamic collaboration between developer and designer with minimum friction and maximum re-use of HTML code. The SMLite engine is designed to be quick, simply and extensible.

Typical design workflow

Lets review a typical situation when the design files are handed over from designer to developer. Design could come in a ZIP file with several .HTML files and subdirectories called "images", "css".

Unpack it all inside templates/default folder of your Agile Toolkit software. Find the HTML file which is more likely to be a base for all the page's and rename it into 'shared.html'. That's all you need to do to create a basic theme for Agile Toolkit.

Your next action would be to outline some of the regions inside this file. It's likely that your designer have already added comments to mark beginning and ending of certain blocks.

The orange dashed line around your page indicates the boundaries of your "Application Template". Unlike other systems (such as Wordpress) you don't create separate header / footer file. Instead the contents are substituted. Inside your template you can outline with the regions two blocks - the header part with images and the actual contents of the page which contains the text. You do not need to delete anything from your template, simply insert the tags. You can add more tags later and as long as they don't overlap, they will not break anything.



Operating with Templates directly

DON'T DO THIS! I EXPLAIN THIS FOR LEARNING PURPOSES.

Similarly to many other systems, you can load up a template and start manipulating it directly. Below is a sample code how you can do it:

```
$tpl = $api->add('SMLite')->loadTemplate('shared');  
echo $tpl->set('Content')->set('Hello World')->render();
```

With this you load the template from shared.html file and replace `<?Content?>` region `<?/Content?>` with the new text. The header part would stay intact and will still contain pictures.

Note that I have replaced the region's content with a static content.

In Agile Toolkit SMLite is used as an underlying tool. You can occasionally use it directly, but most often you would be operating with the template through a View. As a skilled Agile Toolkit developer you should understand how template object is encapsulated inside a view and should be able to understand when is a good time to modify it. It's a good idea to avoid touching templates until you fully read the chapter about Views.

Performance Considerations

Template system in Agile Toolkit is one of the most used pieces of code. Adding even slight feature inside SMLite can have a major impact on the overall performance of Agile Toolkit.

When Agile Toolkit reads a template, it creates a hash-map for all the regions inside a template and breaks down template to a sequence of nested arrays. Any operations with "set" are therefore linear and will not take any additional time even if you operate with huge templates. Rendering template is proportional to the number of tags in the template, but the complexity have a steady growth.

Security Considerations

SMLite is designed for producing HTML code (although it can be used to produce content in different formats too). Therefore when you use `set()` the variable you pass will automatically be encoded. You can specify anything for this method and it will be properly encoded. If you wish to inject a line of HTML inside your template you should use `setHTML()` or `appendHTML()`.

Providing the simplest way to also be the most secure way to specify the content is the policy of Agile Toolkit to improve safety.

AbstractView

The object of AbstractView inherits all of the functionality from AbstractObject, but has some new methods. It also introduces property "template", which contains SMLite object.

defaultTemplate()

When view is initialized, it prepares its default template by creating new SMLite object under "template" property. This template by default is initialized by cloning region "Content" of the owner's template. In the previous section I have described how to create a "shared.html" file, which can then be loaded inside a template. Let's look in more details at how shared.html would look:

```
<html>  
<head>...</head>  
<body>  
  <?Header?>  
  <div class="header">  
      
      
  </div>  
  <?/Header?>  
  <div>  
    <?Content?>  
    <h3>Default Content</h3>  
    <p>Lorem Ipsum Dolor Sit... </p>  
  <?/Content?>  
  </div>  
</body>
```


Let's create a simple view object which would load this template:

```
include 'atk4/loader.php';
$api = new ApiCLI();
$view = $api->add('View',null,null,array('shared'));
$view->template->trySet('Content')->set('Hello World');

echo $view->getHTML();
```

The 4th argument specifies which template to use and because we want our view to load template from a file, we are using `array('shared')`;

Next, let's try to add another view inside our existing view to which we can delegate the work related to making the header look well. For now, let's have the new view clone the `<?Header?>` region.

```
$header = $view->add('View',null,'Header','Header');
echo $view->getHTML();
```

Adding this will have no effect on the output, however you will have a separate view rendering the header recursively. Next I'll create a class object for the header and will use this class instead of "View".

```
class MyHeader extends AbstractView {
    function render(){
        $this->output('Header now works differently');
    }
}

$header = $view->add('MyHeader',null,'Header','Header');
```

Let me clue you in on the arguments to the `add()`. The 3rd argument specifies the spot within the `$view` where the text will appear every time header would call `output()`.

I have made one omission on purpose. My new object outputs without use of the template. Normally a view would work with its own template and then output the data into the parent. Here is the "default" render method:

```
function render(){
    $this->output($this->template->render());
}
```

The 4th argument for `add()` method describes how the template should be initialized. Before we have used `array('shared')` to load template from a file. Specifying string value will point to a region within a parent template, which will be cloned for a child view to use. In our case the default template for the `MyHeader` object would contain: `<div class="header">...</div>`

Alternatively you could also specify both the file and a tag, in which case the template would be irrelevant to the owner's template: `array('shared','Header')`

- `$view->getHTML()`
 - `$view->recursiveRender()`
 - `$header->recursiveRender()`
 - `$header->render()`
 - `$header->template->render()`
 - `$header->output()`
 - `$view->template->appendHTML()`
 - `$view->render()`
 - `$view->output();`

Each view in Agile Toolkit renders itself recursively by calling its `render()` method which then appends the output to owner's template. In the conclusion of this chapter I should note that `Application` class also inherits `AbstractView` and therefore your `$api` also have a template.

ApiWeb and ApiCLI

When building a command-line utility using Agile Toolkit, you are unlikely to use the ApiFrontend - which is an application class for a fully-featured web software. The ApiCLI and ApiWeb are parents of ApiFrontend but they are much more minimalistic:

- ApiCLI provides methods such as dbConnect() and getConfig(), which are useful in command-line applications. This class also initializes some essential system controllers - PathFinder and Logger.
- ApiWeb extends ApiCLI to add proper headers, support for session control, support for stickyGET but also implements main() method. This class will also render all the views you add into it and will automatically use shared.html template. The rendered output is echoed to the browser. This API also implements the concept of Layouts, which is later used in ApiFrontend
- ApiFrontend implements page routing by including page/ *classes or using templates/default/page/templates*. The page is inserted inside <?Content?> region of shared.html file. This class also implements method which is called if page is not found. Finally sub-page functionality is also implemented here.

Which class is right for your application? In most cases, here is a simple way to determine:

- Will your application output HTML? If no - use ApiCLI
- Will your application have more than one page? If no - use ApiWeb
- Will your application use un-usual page pattern? if yes - use ApiWeb
- Otherwise use ApiFrontend.

Use of ApiCLI

Apart from the direct use, you can also use ApiCLI as a cron handler. Because you are using models directly you would be responsible for handling any exceptions yourself.

```
include 'atk4/loader.php';
$api = new ApiCLI('myproj');
$api->dbConnect();
$jobs=$api->add('Model_Job')->addCondition('st','pending');
foreach($jobs as $junk){
    $jobs->set('st','processing')->save();
    try {
        $jobs->process();
        $jobs['st']='done';
    }catch(Exception $e){
        $jobs->recordException($e);
        $jobs['st']='failed';
    }
    $jobs->save();
}
```

In this pattern you will be able to process multiple jobs even if some of them fail and will not run into situation when failed job is retried continuously.

Try not to make the original file too big, create a controller instead which would do things for you.

If you are looking to embed all functionality into a single file, you can declare number of classes after including the loader.

Using ApiCLI as an isolation environment

Because Agile Toolkit is static-safe, you can create ApiCLI from another class and run certain operations or tests inside ApiCLI without impact on the rest of application. This can be done for safety or security considerations, but practically you can create instance of ApiCLI even if you are inside other API.

ApiWeb - Typical use-cases

There are no limitations on how many Application classes you could create. Every time you find a new pattern in your web application development you can create your own application class. Often it's easier to create it on top of ApiWeb instead of

using ApiFrontend, simply because the routing of ApiFrontend would not be relevant.

One design pattern when you would want to use ApiWeb is when you do not want to use page rewriting and front controller. Instead you would want each URL to hit into the target file which would define how it would perform. The situation is for example when you are willing to create a stand-alone URL independent from the rest of your application.

The installer could be a stand-alone application, which should run independently from the rest of your software and perform database migration, add-on installations and more. Using ApiWeb as a basis is quite convenient, because you could direct browser directly to your install.php file.

You can also use ApiWeb for a test-cases. With a separate API most of your application would not be activated and therefore you can focus on testing individual features.

Finally you can use ApiWeb to share fragments of your code with others. It's quite handy to embed classes into a file along with a custom use of API.

Below is example which allows user to specify two numbers and calculates the sum, however the whole implementation is packed into a single file. You can send such a file to anyone who have Agile Toolkit installed and they would simply need to put it into the application folder to make it work.=

(it's a good idea to use schema auto-generation add-on inside your models if you share them)

```
include 'atk4/loader.php';
$api=new ApiWeb();
$api->add('jUI');

$f=$api->add('Form');
$f->addField('line','a');
$f->addField('line','b');
$f->addField('line','result');
$f->addSubmit();

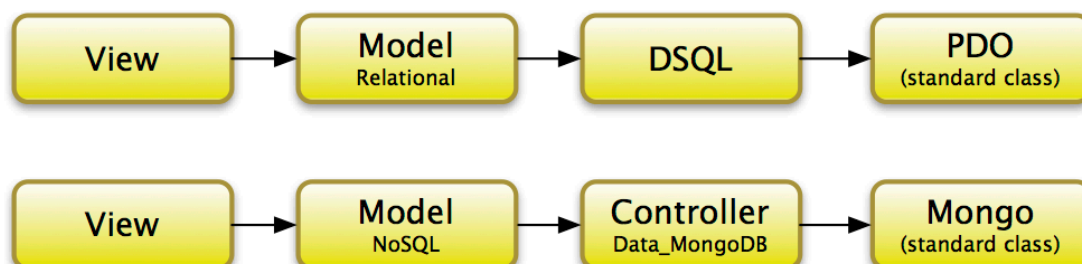
if($f->isSubmitted()){
    $f->getElement('result')->js()->val(
        $f->get('a')+$f->get('b')
    )->execute();
}
```

Creating your own Application Class

When you start your new application with Agile Toolkit you should start by creating your application class. Typically the examples suggest you to create lib/Frontend.php, although you may name it the way you like. If your Application class is located elsewhere you may need to manually include it inside your `index.php`.

Typically the job of an application class is to initialize your application by adding controllers and defining some core logic. Sometimes it's sensible to divide your functionality into two classes - base application class and extended application. This is the best approach if you need to white-label your application or

Data Model



In Agile Toolkit Views can interact with the Data Layer through Models. Model is a class implementing Active Record and ORM as well as holding meta-data for the objects. You must understand that there are two types of models - Relational and Non-

Relational (or NoSQL) and that Model_Table (relational) extends the other. To most of the User Interface both classes are compatible, however both model implementation use a very different way to interact with data.

Before we can understand Relational Models, we must first look into the implementation of the DSQL layer. Models are described in the further chapters.

Connecting to the database

To connect to the database you need to call `api->dbConnect()`; For this method to work without any arguments, you need to specify DSN (connection info) in `config.php` file. Alternatively you can also specify DSN as an argument, although be advised that if you use passwords inside GET argument, they will be visible on exception screen, when your database is unreachable. You can disable error output on the screen, see chapter for "Application" for more information.

Specifying DNS

Using different connection modes

Counting number of queries in application

Using multiple connections

Connecting to noSQL databases

Introduction to Dynamic SQL Object (DSQL)

DSQL is a object aiding in building a query. One object represents one query and collect information about fields, conditions and expressions. A DSQL object may contain other DSQL objects which is used for expressions and sub-selects.

DSQL can convert itself into a PDO parametric query and iterate through results. As a developer it's important that you remember that DSQL will not produce more than one query at a time.

Goals of DSQL

Once query object is created it will accumulate information about SQL conditions, joins, fields, tables. It even stores information about parametric variables and sub-queries. This is important when you need to pass query information from one object to another, between models or even allow controller or add-on to interact with standard query. Multiple queries may exist and can even be executed simultaneously without affecting each-other.

DSQL consolidates all the information necessary for creating a query.

Most ORMs produce query in one go. You can create query in one object then pass it on to another object to add additional queries or joins.

DSQL allows you to add or remove fields, conditions, arguments or anything else at any time. Even after you perform one select, you can re-use the object - extend the query or execute different type of query.

Dynamic Query is grown organically. At any point query can have more fields, conditions or more join added.

When you design queries yourself, you must be aware of differences between different SQL implementations and extensions. DSQL can hide some of the differences between database vendors behind it's object interface.

You create query for all database vendors.

Many ORMs give you a way to use parametric queries, but would not enforce their use. Their limited flexibility often leaves a gap for developers to hard-wire some hand-written queries with a nasty SQL Injection bugs.

DSQL obsoletes SQL language completely and allows you to build ANY query or expression (including select, insert, update, delete, replace and some other queries). If DSQL does not support some type of syntax natively, you can create a new method in DSQL object to handle it.

DSQL should not be used on it's own. DSQL is a tool used by Relational Models and must only be used together with appropriate model.

What features are excluded by design from DSQL

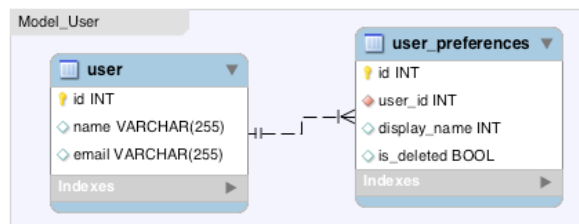
DSQL have no knowledge of your database, tables or table fields. All of the fields must be explicitly specified. DSQL will never change the database structure. DSQL will not have any complex logic apart from query building.

DSQL does not attempt to implement all the possible fetching mechanisms and completely overlap PDO. In fact - you can still access PDO object embedded inside DSQL if you execute care.

Using Multiple SQL tables through Joins

Typical database manipulation systems (ORMs) would strictly bind your object definitions to match your database structure. Some would even look into the database for you and create model. Agile Toolkit allows you to have as many model classes as you need and each can also join as many tables as necessary.

I will start by looking at example with one-on-one relation. Even though the SQL relation is defined as 1:n, our application logic dictates that there will be only one record in user_preference at a time, with is_deleted=0. If we want to present user data and their display_name side-by-side we would need to create a join.



```
class Model_User extends Model_Table {
    public $table='user';
    function init(){
        parent::init();
        $this->addField('name');
        $this->addField('email');
        $pref=$this->join('user_preferences.user_id');
        $pref->addField('display_name');
    }
}
```

My resulting model will still rely on the user table as a primary, and the user_id is used when loading the model, however model will automatically load display_name from user_preferences table, which will appear as a regular field in the model. As a developer working on a back-end you must understand, that model as you define it can be usable by virtually any UI element in Agile Toolkit without any additional tweaks or logic to handle the dependencies. You can purposely define multiple model classes for different use-cases.

Technically "join" in our example returns a new object of class SQL_Join, which can then be used as a proxy to create new field objects. SQL_Join class does not offer you a "addExpression" wrapper, because it wouldn't make a difference for expression anyway. Any fields created will keep reference to a join object. Here is alternative syntax for adding fields:

```
$this->addField('display_name')->from($pref);
```

Presence of this property inside a field will affect it's prefix for the select queries but will also split "update" and "insert" operations into multiple queries. SQL_Join class will automatically determine which ID needs to be used where and will properly alter the sequence of operations.

If you create join with "leftJoin()" method, it will not perform any inserts or deletion to a joined table.

If you are unsure if you should use "join" or "leftJoin", you should probably use "leftJoin"

| defined | insert | delete | update |
|--|-----------------------|-----------------------|-------------|
| <code>\$user->join('preferences.user_id')</code> | user then preferences | preferences then user | both tables |
| <code>\$user->join('address','address_id')</code> | address then user | user then address | both tables |
| <code>\$user->leftJoin('preferences.user_id')</code> | user only | user only | both tables |
| <code>\$user->leftJoin('address','address_id')</code> | user only | user only | both tables |

leftJoin actually defines cascading preferences, but if you looking to use different type of joins inside select query, you can specify 3rd argument to the method. You can probably notice the similarity with `$dsql->join()`.

You can also use on-to-one joins with `join('pref.userId','userId')`; but Agile Toolkit would still attempt to insert into one table first, then another.

Dealing with Duplicate Fields

Sometimes you would have the field with the same name present in both tables. Agile Toolkit does not permit multiple fields with same name within a model, so you would need to specify an actual field. This can be done with second argument passed to `addField()`; It works in the same way is with model fields.

```
$pref->addField('is_pref_deleted','is_deleted')
->type('boolean');
```

Fundamentals of DSQL

To understand DSQL you should realize that it works in a very similar way to how other objects in Agile Toolkit working. This consists of two parts:

1. Query parameters are set and are saved inside a property.
2. Query is rendered, parameters are converted and query is executed.

Here is the example of a sample query template:

```
select [myexpr]
```

DSQL can have only one template at a time. Calling some methods will also change the template. You can explicitly specify template using `expr()`.

Before query is executed, DSQL renders the template. It searches for the regions in brackets and executes method `render_`

```
function render_myexpr(){
    return 'now()';
}
```

now to try and put this all together you would need to use this code:

```
class MyDSQL extends DB_dsql {
    function render_myexpr(){
        return 'now()';
    }
}

$this->api->dbConnect();
$q = $this->api->db->dsql('MyDSQL');
$q->expr('select [myexpr]');
echo $q->getOne();
```

In this example, we define our own field renderer and then reference in the expression. The resulting query should output a current time to you.

DSQL Argument Types

When you build the query by calling methods, you arguments could be:

1. Field Names, such as `id`
2. Unsafe variables, such as :a
3. Expressions such as "now()"

```
$q=$this->api->db->dsql()->table('user');
$q->where('id',$GET['id']);
$q->where('expired','<',$q->expr('now()'));
echo $q->field('count(*)')->getOne();
```

The example above will produce query:

```
select count(*) from `id`=123 and `expired`<now();
```

Value from \$GET[id] is un-safe and would be parametrized to avoid injection. "now()" however is an expression and should be left as it is.

It is a job for the renderer to properly recognize the arguments and use one of the wrappers for them. Let's review the previous example and see how to use different escaping techniques:

```
class MySQL extends DB_dsql {
    function render_test1(){ return 123; }
    function render_test2(){ return $this->bt(123); }
    function render_test3(){ return $this->expr(123); }
    function render_test5(){ return $this->escape(123); }
    function render_test6(){ return $this->consume(123); }
}
```

Those are all available escaping mechanisms:

1. bt() will surround the value with backticks. This is most useful for field and table names, to avoid conflicts when table or field name is same as keyword ("select * from order")
2. expr() will clone DSQL object and set it's template to first argument. DSQL may surround expressions with brackets, but it will not escape them.
3. escape() will convert argument into parametric value and will substitute it with sequential identifier such as :a1, :a2, :a3 etc.
4. Finally, consume() will try to identify object correctly and adds support for non-SQL objects such as Model Fields.

Exercise

This exercise is only for your understanding. For normal usage of DSQL you don't need to do this.

Let's review how the "set" operator works in "update" query. First, let's use the following simplified template:

```
update [table] set [set]
```

To help users interface with our template, we must have the following two methods:

```
function table($table){
    $this->args['table']=$table;
    return $this;
}
function set($field,$value){
    $this->args['set'][]=array($field,$value);
    return $this;
}
```

It's important that you don't use any escaping mechanisms on the arguments just yet. They may refer to expression which can still be modified from outside. The arguments are packed into an internal property "args". Next, let's review the rendering part of

the arguments. This time I'll be using different escaping mechanisms in different situations.

```
function render_table(){
    return $this->bt($this->args['table']);
}
function render_set(){
    $result=array();
    foreach($this->args['set'] as list($field,$value)){
        $field=$this->bt($field);
        if(is_object($value)){
            $value=$this->consume($value);
        }else{
            $value=$this->escape($value);
        }
        $result[]=$field.'='.$value;
    }
    return join(' ', $result);
}
```

Table would need to be back-ticked and we don't really need to worry about expressions. For the "set" rendering things are bit more complex. We allow multiple calls to set() and then we need to produce the equation for each field and join result by commas. The first argument, the field, needs to be back-ticked. Second argument may be an object, but if it's now, it most probably would contain an un-safe parameter, so we use escape() to convert it into parametric value.

Consume here would recursively render the expression and join the parameters from the subquery to our own. In some situations we would need to surround consume with a brackets, if SQL syntax requires it.

NOTE: This exercise is mentioned here only to help you understand DSQL. You must not try to re-implement rendering of table or set arguments as it is already implemented properly in DSQL. Let's take look at other templates DSQL supports.

Direct Use of DSQL

DSQL Modes

With DSQL you can set arguments first and then decide on the template for your query. DSQL recognizes a number of SQL query templates:

```
'select'=>"select [options] [field] [from] [table] [join] [where] [group] [having] [order] [limit]"
'insert'=>"insert [options_insert] into [table_noalias] ([set_fields]) values ([set_values])"
'replace'=>"replace [options_replace] into [table_noalias] ([set_fields]) values ([set_value])"
'update'=>"update [table_noalias] set [set] [where]"
'delete'=>"delete from [table_noalias] [where]"
'truncate'=>'truncate table [table_noalias]'
```

To render the templates, all the respective renderers exist (such as render_options, render_field, render_from, etc). Data is typically stored in property "args" until execution. Below is the list of functions you can call on the dsql() object to affect the queries:

- `option("ignore")`
- `field($field, $table, $alias)` // uses multiple call formats
- `table($table, $alias)` // uses multiple call formats
- `where($field,$condition,$value)` // uses multiple formats
- `join($table, $master_field, $join_type)` // uses multiple formats
- `group("field")`
- `order($field, $desc)` // uses multiple formats
- `limit($count,$offset)`

- `set($field,$value)` // used by insert, replace and update to set new field values

When using any of the mentioned templates, the property "mode" of dsql object will be set to the name of the template (such as 'update' or 'insert').

Properties of DSQL Object

I have already mentioned property "template", "args" and "mode", but there are more properties. All of them are defined public and you can access them for reading, but avoid changing them.

- `params` - During rendering of the query, PDO parameter values are stored here.
- `stmt` - Contains PDO object, in case you need to use some custom PDO values or properties
- `main_table` - When using joins, this contains the name (or alias) of main table, which is sometimes prefixed to the fields.
- `default_field` - Contains name of the field to use if you don't set any fields. Asterisk by default.
- `default_exception` - will throw `Exception_DB` by default
- `param_base` - prefix used when creating params for PDO
- `debug` - can be switched on by calling `debug()` method. Will echo query before executing it.
- `sql_templates` - contains array with pre-defined modes and templates
- `id_field` - can be set if you wish that DSQL would return id as key when iterating.

Cloning and duplicating DSQL

Any DSQL object can be safely cloned. It will NOT however clone any arguments.

Alternatively you can call `dsql()` method, which will create new instance of the same class, very similar to `newInstance()`.

If you are using `expr()`, it will return a new object with the template as you specify in the argument. This is done to leave your original query intact. If don't want new object, `useExpr()` will change template without cloning.

Overloaded arguments

Similarly how you could specify different argument types is jQuery methods, you can specify different argument types to the methods. Keep your code clean by using most effective one, e.g. use `where('id',123)` instead of `where('id','=',123)`

Specifying Table

Use this argument to set a primary table or even multiple tables for your query. When specifying multiple tables, you would most likely need to have where condition. Call like this:

- `table('user')`
- `table('user','u')` // aliases table with "u"
- `table('user')->table('salary')` // specify multiple tables
- `table(array('user','salary'))` // identical to previous line
- `table(array('u'=>'user','s'=>'salary'))` // specify aliases and tables

Getting field with table prefix

When a single table is used, you can use method `getField($field)` to generate expression with appropriate table prefix and back-ticked field. Please do not confuse this with `getElement()` or `model->getField()`;

```
$q=$this->api->db->dsql();
$field = $q->table('book','b')->getField('id');

// will contain expression for: `b`.`id`
```

Specifying fields

Can add physical fields or fields expressed with formula (expressions) to the query. Can also alias fields. Formats:

- `field('name')`
- `field('name','user')`
second argument specified table
- `field('name','user')->field('line1','address')`
- `field('name,surname')`
multiple fields in one go
- `field(array('name','surname'))`
can specify as array
- `field(array('n'=>'name','s'=>'surname'))`
use aliases for fields
- `field(array('n'=>'name'),'user')`
combine explicit table and field alias
- `field($q->expr('year(now)-year(birth)','age'))`
alias expression
- `field(array('age'=>$q->expr('year(now)-year(birth)')))`
multiple expressions
- `field($q->dsql()->table('book')->field('count(*)','books'))`
subquery
- if you don't call `field()` at all, "" will be selected.

Adding Conditions (where)

Methods we have reviewed till now in details have been using "safe" arguments. When you are using `where()` you can use unsafe argument. While we still advise you to perform input validation from `$_GET()`, but in normal circumstances it's quite safe to use those as the last argument to `where`.

Below are execution examples:

- `where('id',1)`
argument 1 here is unsafe, will use parametric variable
- `where('id','>',1)`
can use any operation. Second argument must be safe
- `where('id>',1)`
same as above, but only limited number of operators are supported: <,>,<=,>=,<>,! =,in,not in,like,is and spaces
- `where('expired','<',$q->expr('now()'))`
expression overrides unsafe variable, will not use param
- `where('age',$q->expr('between 5 and 18'))`
can be used with safe variables
- `where('age',$q->expr('between [min] and [max]')->setCustom('min',$min)->setCustom('max',$max))`
use this with un-safe variables
- `where('book_id',$q->dsql()->table('sale')->field('book_id'))`
subquery
- `where('state',array('open','closed','resolved'))`
state in ('...!...')
- `where($q->expr('length(password)'), '>', $maxlen)`
expression

When you are calling `where()` multiple times like this, all the conditions are joined using "and" clause. DSQL also supports OR, but multiple OR conditions are considered a single `where()` condition. For example:

```
select user where is_active=1 and (id=1 or id=2)
```

When you look at the query like this, you should see that area in brackets is actually an expression. A method `orWhere()` can help you produce this expression:

- `where('is_active',1)->where($q->orWhere()->where('id',1)->where('id',2))`
- `where('is_active',1)->where(array(array('id',1),array('id',2)))`

same, shortcut

- `where('is_active',1)->where(array('name is null','surname is null'))`

The second and third format here would use `orWhere()` internally, but possibly is a little bit more readable (especially if you write it on multiple lines).

Finally if you want to compare two fields, you can use something like this:

- `where('income','>',$q->expr($q->bt('expense'))`
`where `income`>`expense``
- `where('income','>',$q->getField('expense'))`
better one. Use this.

The second format here will also use a table prefix for expense, which is nice when you are using multiple joins. You can also use `getField` for the first argument:

- `where($q->getField('income'),'>',$q->getField('expense'))`

Finally, you can also use `andExpr()`, which works identical to `orWhere()`.

```
$q=$this->api->db->dsq1()->table('book');
$q->where('approved',1)->where($q->orWhere()
    ->where('state','submitted')->where($q->andExpr()
        ->where('state','pending')->where('filled',1)));
```

Resulting query:

```
select * from book where `approved`=1 and
    (`state`='submitted' or
    (`state`='pending' and `filled`=1)
    )
```

Using having()

The "having()" method is described as an alias to `where()` except that the result for all `having()` clauses will be stored in a separate location and will render under a separate tag. Anything written above for `where()` also apply for `having()`.

Joining tables join()

With this method, you can create all sorts of joins and query data from multiple tables. Join will use default field names, which follow the following principle: "If you have "book" and "author" table, joining them would use `book.author_id=author.id` expression."

I advise you to use this field naming convention in your database design, which will simplify your code and make it more readable. The examples below, assume that you already have a query object and the primary table is set:

```
$b=$this->api->db->dsq1()->table('book');
```

- `join('author')`
`join author on author.id=book.author_id`
- `join(array('a'=>'author'))`
same, but sets alias to `a`
- `join('chapter.book_id')`
`join chapter on chapter.book_id=book.id`
- `join(array('c'=>'chapter.book_id'))`
same but sets alias to `c`
- `join(array('a'=>'author','c'=>'chapter.book_id'))`
two joins

If you are not using the standard convention you can use second argument to specify the field in your main table (book):

- `join('author','written_by')`
join author on author.id=book.written_by
- `join('info.issn','issn')`
join info on info.issn=book.issn

If you need a further flexibility with the join, you can use expressions. When combined with `orExpr` / `andExpr` it gives you quite powerful tool:

- `join('info',$q->andExpr()->where('info.issn',$q->getField('issn')->where('info.is_active',1))`
join info on info.issn=book.issn and info.is_active=1

The third argument can be used to change the join type. It is an unsafe argument.

- `join('info',null,'inner')`
inner join. By default "left" join is used

Using grouping (group())

Group specifies how the results will be grouped (group by):

- `group('sex')`
- `group($q->expr('year(created)'))`
- `group('sex,age')`
- `group(array('sex',$q->expr('year(created)')))`

Ordering results (order())

Ordering can be done my one or several fields and in two directions.

- `order('age')`
- `order('age','desc')`
reverse order
- `order('age',true)`
same as above, reverse oredr
- `order('age,sex')`
order by age, sex (multiple fields)
- `order(array('age','sex'))`
- `order('age desc, sex')`
- `order(array('age desc','sex'))`
- `order($q->expr('year(birth)'))`

Options

There are two methods for setting options:

- `option('calc_found_rows');` // option for select
- `option_insert('ignore');` // option for insert

More option methods could be added later.

Using setCustom

If DSQL template encounters a tag which it can't find method to render, it will next check `args['custom']` to see if it might have been set directly. This allows you to set a custom values for the expression tags. If you use one of the pre-defined tags, then your value will override the default rendering mechanism. `addCustom()` uses `consume()` escaping on it's argument.

```
$q=$this->api->db->dsq1()->expr('show table [mytag]');
```

- `addCustom('mytag','123')`

```

    produces 123
    • addCustom( 'mytag', $q->bt( 'user' ) )
    produces `user`
    • addCustom( 'mytag', $q->expr( 'foo bar' ) )
    produces foo bar
    • addCustom( 'mytag', $q->escape(123) )
    produces :a1 param

```

Deleting arguments

By calling method `del()` you can get rid of some arguments which have been set before:

- `field('bleh')->del('fields')->field('name')`
select name from ..
- `reset()`
will delete all arguments

The most popular use is when you need to change fields on arbitrary query:

```

$y=clone $q;
$count = $y->del( 'fields' )->field( 'count(*)' )->getOne();

```

Calling Methods (`call()` and `fx()`)

When you call your own methods in SQL the typical syntax is:

```
call myfunc(args)
```

To do this with DSQL, you should use the following pattern:

```

$q=$this->api->db->dsq();
$q->call( 'myfunc', array($arg1, $arg2));

```

To call standard functions you can use `fx()` method.

```

$q=$this->api->db->dsq();
$q->call( 'myfunc', array($arg1,
    $q->dsq()->fx( 'if', array($x1,$x2,$x3) )
));

// call myfunc($arg1, if($x1,$x2,$x3))

```

Note: `call()` and `fx()` will not clone your query, so use `dsq()` if necessary.

Using `set()`

You should use `set()` for insert and update. In fact, you can use something like this:

```

$q=$this->api->db->dsq();
$q->set( 'name', 'John' )->set($user_data);
$q->where( 'username', $username )->limit(1);
if(!$q->get()){
    $q->insert();
}else{
    $q->update();
}

```

WHY YOU SHOULDN'T BUILD YOUR QUERIES WITH DSQL

Within DSQL you might find the comfort and control over SQL queries, but you must try to avoid using DSQL directly. Relational Models are higher level objects which give you much simpler syntax with very minimal performance hit.

Cross-SQL compatibility methods and shortcuts

Gradually, as Agile Toolkit starts supporting more database engines, it will implement more compatibility methods. Currently the following methods are present:

- `concat (arg, arg, ...)`
string concatenation
- `describe (table)`
select table schema
- `random ()`
return random value 0..1
- `sum (field)`
returns expression for sum(..)
- `count (field)`
returns expression for count(..). Argument is '*' if unspecified.

Query Methods

Calling any of the methods below, will execute your query. Those methods take no arguments, but will render query collecting previously specified data.

- `execute ()`
will not change template. Use with your own expression.
- `select ()`
- `insert ()`
- `update ()`
- `replace ()`
- `delete ()`
- `truncate ()`

Additionally there are some combinations of executing query and fetching data.

- `get ()`
returns array of hashes containing data
- `getOne ()`
returns scalar value, first row first column
- `getRow ()`
return first row only, but as non-associative array
- `getHash ()`
return first row as a hash
- `fetch ()`
returns one row as hash. Calling this or previous two methods multiple times will return further data rows or null if no more data is there.

Iterating through results

You can use DSQL as an iterator:

```

$q=$this->api->db->dsql();
$q->table('book')->where('is_active',1);
foreach($q as $data){
    $qq = clone $q;
    $qq->where('id',$data['id'])
        ->set('data',json_encode(..))
        ->update();
}

```

Using preexec

Normally, while you build the query you can set an options, but the query is only executed when you start iterating through it. If you wish to execute query quicker you can use `preexec()` method. This allows you to have some information about the query even before you start iterating.

You can also execute multiple queries simultaneously for better performance.

```

$q=$this->api->db->dsql();
$q->table('book')->where('is_active',1);
$rows = $q->calc_found_rows()->preexec()->foundRows();
foreach($q as $data){
    $qq = clone $q;
    $qq->where('id',$data['id'])
        ->set('data',json_encode(..))
        ->update();
}

```

In this example, we were able to determine number of `$rows` in result set before we started iterating through the selected cursor and avoided executing query twice.

Debugging

You can use `$dsql->debug()` to turn on debugging mode for the query. This will output some data using "echo" when the query is being rendered. DSQL will use font colors / HTML to highlight different parts of query.

DSQL - Practical Examples

- Book (title, author_id, is_published)
- Chapter (book_id, no)
- Author

I will review number of tasks

```

$db = $this->add('DB')->connect($dsn);

// Get All Books
$q = $db->dsql()->table('books');
$books = $q->get();

// Get all books of particular author
$book_by_author = $q->where('author_id', 1)->get();

// Delete all books by particular author
$q->delete();

```

Compared to other query builders - DSQL is more fluid, allows calling methods in any order and has a straightforward syntax.

```
$q = $db->dsql()->table('user');
$q->where($q->orExpr()
    ->where('id',1)
    ->where($q->andExpr()
        ->where('age','>',25)
        ->where('votes','>',100)
    )
);
$users = $q->get();
```

```
$users = $db->dsql()->table('user')
    ->join('phone.user_id')
    ->field('users.email,phone.number');
```

```
$q = $db->dsql()->table('user');
$q->join('phone',$q->orExpr()
    ->where('user.id=phone.user_id')
    ->where('user.id=phone.contact_id')
);
$users = $q->get();
```

```
$min_age = $db->dsql()->table('user')
    ->field('min(age)')->do_getOne();
```

```
$q=$db->dsql()->table('users');
$q->set('votes',$q->expr('votes+1'));
$q->update();
```

Introduction to Models

In our software data is arranged into logical tables or collections. It's sensible to associate data with a set of relevant actions. PHP objects are an ideal container for both the data and the set of methods which can be applied on the data.

Normally a single model object represents a single row of data, however this approach is impractical. Agile Toolkit uses a different approach for the models, where the model can load any row of data or even perform operations on all data set. If you have prior experience with other data frameworks such as Doctrine, Active Record or some other ORM, you should pay attention to how "Model" works in Agile Toolkit.

The Dataset

Models in Agile Toolkit are not limited to SQL, but can in fact work with many different data drivers. Therefore I will define all the possible data you can access through one model as a Dataset.

Dataset is determined by 3 things: 1) Driver 2) Table 3) Scope.

| Driver | Table | Condition |
|-----------------------|-----------------|--|
| SQL + Database/Schema | Table Name | set of "where" conditions joined by AND clause |
| Memcache | Key Prefix | Sub-prefix |
| MongoDB | Collection Name | Conditions |
| Redis + Object Type | Object name | Prefix |

Relational Model

A significant segment of the database implementations are so called RDBMS - Relational Database Management Systems. Notable for their flexibility in data querying they utilize a standardized query language - SQL. Agile Toolkit takes advantage of the powerful features of RDBMS (joining, sub-selects, expressions) and has a significantly enhanced model class to work directly

with the database through DSQL.

You can find a detailed description of relational models further in this book. Even though the relational models are significantly enhanced, they still retain the functionality of regular models, so everything described in this chapter would also apply to relational models.

setSource - Primary Source

A non-relational models can use setSource() method to associate themselves with a driver. Driver is an object of class extending Controller_Data. Model will route some of the operations to the controller, such as loading, saving and deleting records.

Model can only have one source and because relational models already using SQL you cannot specify a different source.

addCache - Caches

A single model can have several caches associated with it. For example a relational model may have Session cache.

When loading model with associated cache - the first attempt is made to load the model from the cache directly. If model is not found in cache(s), the primary source is used as a fall-back.

When saving model data, it will be also saved into all the associated caches.

The data controllers typically can be used as either primary source or as a cache.

Model data and methods

In a typical ORM implementation, model data is stored in model properties while reserving all the property names beginning with underscore. Agile Toolkit stores model data as array in a single property called "data". To access the data you can use set(), get() or array-access (square brackets) format.

Before you can access the data, however, you must define some fields. Below is a typical implementation of a model in Agile Toolkit. Please note that model is defined using PHP language and it's always defined as a class.

```
class Model_User extends Model {
    function init(){
        parent::init();
        $this->addField('name');
        $this->addField('surname');

        $this->addField('daily_salary');
        $this->addField('due_payment');
    }
    function goToWork(){
        $this['due_payment'] = $this['due_payment']
            +$this['daily_salary'];
        return $this;
    }
    function paySalary(){
        echo "Paying ".$this['name']. " amount of ".
            $this['due_payment'];
    }
}

$m=$this->add('Model_User');
$m['name']='John';$m['daily_salary']=150;

for($day=1;$day<7;$day++) $m->goToWork()
$m->paySalary();
```

As you see in the example, model User's model combines definition of the fields with the methods to perform business operations with the model. When you design model methods, it's important that you follow these guidelines:

- Never assume presence of UI.
- Avoid addressing "owner" object.
- Keep object hierarchy in mind. Extend "User" model to create "Manager" model.
- All field names must be unique

By following these guidelines, you can design a model which can work with magnitude of data sources.

Loading and Saving models

You can save your model data to a primary source driver or load data if you know the "id" of the record. The "id" is not necessarily a number, but it uniquely defines a data within source / table.

Let's extend our user model by adding "Session" source.

```
class Model_User extends Model {
    public $table='user';
    function init(){
        parent::init();

        $this->setSource('Session');
```

Once source is set, you can use a number of additional operations:

```
$m['name']='John';$m['daily_salary']=150;
$m->save();
echo $m->id;    // will contain a generated ID

$m->load($other_id);    // load different record into model
```

Model objects in Agile Toolkit are not tied in with any particular record. They can load any (but one) record from the data-set and save it. A single object can also iterate through the data-set by loading each individual record.

There are only two properties which are affected when you load model: "data" and "id". Next example demonstrates how to display list of all the users and their respective "due_payment" field:

```
foreach($m as $row){
    echo "Please pay ".$row['daily_salary']." to ".
        $row['name']."\n";
}
```

When iterating, the \$row becomes automatically associated with the "data" property, however if you are willing to change the content of the model, you should use the \$m instead:

```
foreach($m as $row){
    $m->paySalary();
}
```

Model's method loaded() will return true if model have been loaded with any data from the source and false otherwise.

```
$m=$this->add('Model_Table');
$m->loaded();    // false
$m->load(1);
$m->loaded();    // true
$m->unload();
$m->loaded();    // false
```

Deleting model data

You can delete a single record of data by calling delete(\$id) method or you can remove all data by calling deleteAll(). If you do not pass \$id to delete() method, then the currently loaded data will be deleted.## Introduction to Relational Models of Agile Toolkit (ORM)

Relational Model base class (Model_Table) extends standard models and enhances them with various features. If you are already familiar with other ORM (Object Relational Mapper), you might wonder why it was necessary for Agile Toolkit to have its own Model implementation.

The fact is that Agile Toolkit is incredibly agile in the way how you are able to

Goal: Unified factory and entity

In a traditional design, you require to have a factory object (or class) which will produce objects for each data record. As I have already mentioned in the previous chapter, the Model implementation in Agile Toolkit unifies both purposes together. The primary reason for this decision is the nature of a short-lived PHP requests. While the separate class design is better for threaded Java, in PHP you would need to create two classes per model on every request.

Unifying everything into a single model comes at a cost and that's why a single model object is allowed to load different data entries. The same principle applies to the relational model.

Goal: Simplification of SQL

As you describe the model, it will behind the scenes build the DSQL object which can be used for accessing, saving and deleting your model data. The nature of DSQL which allows it to perform multiple queries perfectly matches the needs of reusable models. At the same time, the power and flexibility of DSQL can still be accessed, if you want to optimize your queries.

In typical ORM design, you must either use their limited features or use completely different way to address database. For example, you normally are unable to perform action on multiple records through ORM interface, and if you wish to do so, you would need to come up with a full query yourself.

Agile Toolkit models can provide you with a pre-made DSQL object for you which you can extend and modify.

Goal: Integrity of DataSet

Relational models perfectly applies the concept of data-set to your database. You can define multiple models which access the same table, but would have a different set of conditions and therefore would have different data-sets.

Agile Toolkit ensures that you would not be able to accidentally go outside of the data-set when you query, update, delete or insert data.

How Agile Toolkit ORM is unique?

ORM (Object Relational Mappers) are software API layers which are designed to abstract data manipulation backend and create a simpler interface. If you have used ORMs you would know the downsides:

- Typical ORM have Exponential Complexity. Initially they offer a simpler way to access data, but as the demand of the application grows, you end up hacking around the ORM limitations to a point where it will become complex and un-stable.
- Most of ORM models use either XML or YAML to define data model and map tables against classes. As a problem you give up on Object-Oriented approach and the whole flexibility given to you by a Query language.
- Typical ORM implementations have a set of fields they go by and are rather inflexible about accessing data selectively.
- Typical ORMs would only allow you to work with the defined models. You cannot change model definition dynamically.
- Schema refactoring affect existing models of ORMs and therefore can break your working code. Manually extending SQL is often very difficult with ORMs.

Agile Toolkit ORM solves ALL of the above problems elegantly

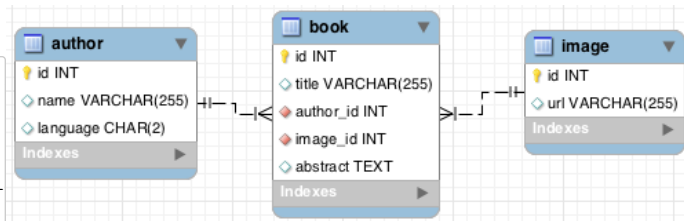
Complexity control

Imagine the following data scheme:

The typical ORM would create 3 classes, one class for each table and will also detect relationship between the tables to allow traversal. Now suppose, you need to show a page, which would contain titles of all the books by some author. Here is a typical

way to do it with GENERIC ORM:

```
$author = authorfactory::find($_GET['id']);
$books = $author->books;
foreach($books as $book){
    $url = $book->image[0]->url;
    // use $book->name / $url to display
}
```



The code seems simple enough, but unfortunately it generates large amount of queries: line #1 loads author and creates a query. line #2 traverses into books and also executes a query. It may also fetch all the books into memory (and it will include book abstracts, which is a large field, even though we might need only limited number of records. #4 traversing into "image" table would create one request per record creating a total of 1 simple query, 1 query with large result set and N more queries (for each result). If you know SQL, then you can write same thing in a single query:

```
select name, url from book join image on image.id=book.image_id join author on author.id=author_id limit 10
```

Agile Toolkit ORM is much more efficient with the queries while retaining a simple object-oriented syntax. And I must add, that you do not need to define joins inside your "book" model for it to work (as you might not need them there all the time).

```
$author = $this->add('Model_Author');
$author->addCondition('id', $_GET['id']);
$books=$author->ref('Book');
$books->join('image')->addField('url');
foreach($books as $row){
    // use $books['name'] and $books['url']
}
```

The above code will produce the exact same query with two joins, even though the initial definition of books wouldn't have those joins defined.

Agile Toolkit will not perform queries when traversing through SQL models

Refactoring Benefits

You might think by now, why write 5 lines of Agile Toolkit ORM code if the same thing could be done by crafting a single SQL query? The benefit is the refactoring.

CASE1: Suppose our system-wide "Model_Author" must now be limited only to english authors. In Agile Toolkit you would modify init() method of the model to add addCondition('language','en'); As a result the query crafted for fetching books will also be affected appropriately throughout all the project.

If you would have used SQL code, you would need to go through all the queries where you somehow reference author table and double-check your condition. The typical case when this is a big problem is if you decide to implement "soft-delete".

CASE2: Suppose that you no longer wish to store image's URL but you would rather calculate it on the fly. Therefore you are looking to remove the field from the database and replace it with SQL expression.

If you are building your own SQL queries, then you would need to change all of your code throughout the project and copy-paste the exact expression into all the queries. Agile Toolkit allow you to define expressions as model fields and then re-use them. Here is a slightly updated code, which will perform a sub-select on the "image" table instead of join:

```

$author = $this->add('Model_Author');
$author->addCondition('id',$_GET['id']);
$books=$author->ref('Book');
$books->addExpression('url')->set(function($m){
    return $m->refSQL('image_id')->fieldExpr('url');
});
foreach($books as $row){
    // use $books['name'] and $books['url']
}

```

Now the Book will use model for the "Image" to retrieve the expression for URL no matter how complex it is. Naturally, if URL is just a table field it would also work fine.

The important benefit here is contingency of queries.

Object-Oriented Models

As you have already learned in the previous chapters, Model definition in Agile Toolkit is done through object-oriented classes. Therefore you can easily create a dedicated model for your type of query:

```

class Model_AuthorsBook extends Model_Book {
    function init(){
        parent::init();
        $this->join('author')
            ->addField('author_lang','language');
        $books->addExpression('url')->set(function($m){
            return $m->refSQL('image_id')->fieldExpr('url');
        });
    }
}

```

While typical ORM associates Models with Tables, Agile Toolkit associates Models with Queries

Actual Fields

Models of Agile Toolkit will query all model fields by default. You can however use the `setActualFields()` method to specify a restricted set of fields which you need for your query.

There is another significant advantage of the way how models are defined with Agile Toolkit. Suppose you add a new join between a "book" and "book_info" table. It's possible that if you update your "Model_Book" with the join, it would affect your existing code. Therefore if a large amount of code already depends on this model, you can extend it into "Model_JoinedBook", which implements the join. This approach will leave all your stable code un-changed.

Manual Intervention into SQL

While most ORM would rely on some internal mechanics for building the actual query - Agile Toolkit relies on DSQL. And since you are already familiar with this object, you can now extend it in any way you like to move complexity from PHP into SQL. For example, in order to add a field which would contain number of books each author have, you would need to add this line into `init()` method of `Model_Author`:

```

$this->addExpression('books_written')->set(function($m){
    $dsq = $m->refSQL('Book')->count();
    return $dsq;
});
$this->addExpression('full_name')->set(
    $this->dsq()->concat(
        $this->getElement('name'),
        ' ', $this->getElement('surname')
    ));

```

As a bonus, I also added expression for "full_name".

Conclusions

Relational Models in Agile Toolkit are able to use PHP language to describe complex SQL queries but also abstract tables, fields, subqueries and conditions. This is all done with minimum overheads on the PHP side.

Advantages of using Agile Toolkit ORM are overwhelming. Consistency, stability, security, extensibility, abstraction are only few of the benefits. Before you could understand how to use Agile Toolkit ORM most efficiently, I must take you to one more diversion to introduce you to Model Fields.

Relational Model Fields

Model uses objects to describe each of its field. Therefore when you create a model with 10 fields, it would populate at least 12 objects: Model's object. One object for each field. DSQL object for a model in general. Field object plays a very direct role in creating queries for the model such as select, update and insert queries.

Normally you would rely on the standard Model Field class, which is populated when you call `addField()`, however you should know that it's possible to extend fields significantly.

Field Meta-information

One of the significant roles carried out by Model Field is collecting of the information about the field. Information such as caption, type, possible values, associated model is stored in Field's properties and can be extended to hold even more information such as caption, validation rules etc.

Query Generation

Model will operate with internal DSQL object. It is however responsibility for the field object to populate queries with field information.

Basic Use

You would typically create a new field by calling `addField($name)` method. This method is a shortcut for “return `add('Field',$name);`” but should be used for standard field definition.

```
$this->addField('name');
$this->addField('gender')->enum(array('m','f'));
$this->addField('language')->setValueList($languages);
```

There are several classes which extend “Field”, such as:

- `Field_Reference` - implements one-to-many relation field
- `Field_Expression` - implements a custom SQL query readonly field
- `Field_Deleted` - implements field used for soft-delete
- `filestore/Field_File` - implements field for referencing uploaded files
- `filestore/File_Image` - implements field for images and thumbnails

Some of them will automatically be used by the Model when you call a dedicated method (`addExpression()`, `hasOne()`), others can be manually added using `add()` method.

The `short_name` of the field object corresponds to the field name and will be used for internal referencing as well as in result sets. If you pass second argument to `addField()`, then you can specify a different actual database field.

```
$this->addField('name','full_name');
$field = $this->getElement('name');
```

Anything what was said in the `AbstractObject` section still applies to the field, for example - you can remove fields from model: `$field->destroy();`

Relational Model Fields and DSQL

Since fields are associated with models and models know which table they work with, you can use Fields of Relational Models inside SQL expressions.

```
$model->dsql()  
->set($model->getElement('name'),'John')  
->update();
```

It's highly advisable that if you use field of a model \$model, you should use the query from \$model->dsql();

Definitions: type

Several methods of the field are acting to describe model type.

- `type()`
will specify the primary type of the field. Default value is text, but could also be: int, date, money, boolean. Views will attempt to properly display recognized fields, for example, date will be converted to a regional format on Grid and will use date-picker inside a form.
- `display()`
specifies the exact field / column class to use for this field. When Form displays fields it relies to objects extended from Form_Field. By specifying value here, you can define how to display the value.
`$model->addField('pass')->display('password');` If you specify the array, you can define the particular value for each view: `display(array('form'=>'password', 'grid'=>'text'));` You may also specify fields from add-ons in the format of 'addon/field_name'.
- `allowHTML()`
can be set to "true" if you want field to be able and store HTML. By default forms will strip HTML from the input for security. If you wish to display HTML in-line, you might need to use setHTML() method of the template engine.
- `mandatory()`
setting this to true will present the field as a mandatory on forms.
- `defaultValue()`
specified value will be set to the field by default.

You must remember, that properties of a model serve the purpose of configuring Views to comply with your configuration. They will not restrict or validate the actual model. For example - you can still create model entry with a empty mandatory field.

If you wish to validate fields on the model level, you will need to use beforeSave hook, this will have a system-wide access regardless of who is accessing the model. Even if you create a validation like that, you can still use model's DSQL to manually create a query which would bypass the validation.

Actual Fields

Typically there are many more fields in a model than you would need to display on the forms. With actual fields, you can specify which fields you need to display to user and in which order.

While actual fields primarily is a functionality of a respective view (such as defining which columns are visible inside grid), it also has impact on the query. Agile Toolkit models will never use "" to load model data.

```
$page->add('Grid')  
->setModel('User',array('name','email'));
```

This code will display grid with only two fields in exactly the specified order. Not always you would want to specify a list of fields. If field list is omitted, then model will attempt to determine which fields to display based on number of flags.

- `system()`
field will be loaded by model always, even if not present in actual fields. Setting system to true will also hide the field, but it can be un-hidden.
- `hidden()`
field does not appear in the grid, form etc unless specifically added or requested by actual fields

- `editable()` / `visible()`
overrides hidden but for only particular widgets. E.g. if field is hidden, but editable it will appear on the form, but not in grid. Setting `editable(false)` will hide field from form even if it's not hidden.
- `readonly()`
the field will appear on the editable form but will be displayed using read-only field.

These methods can accept "true", "false" or "undefined" value.

Grid-related attributes

Several attributes change how fields are displayed inside a grid or filter:

- `searchable()`
makes field visible in filter form.
- `sortable()`
turns on sorting control on the grid column.

Value lists and foreign keys

Some fields contain one value, but the value needs to be displayed differently. For example it's typical to have 'm' for Male and 'f' for Female.

- `listData()`
Specify associative array used to decode the value of the field. Inside a Grid, the key value will be substituted with the string: `array('m'=>'Male')`; Inside a form, the field will be presented as a drop-down.
- `enum()`
specify array (without keys) if you only want a drop-down but do not want value substitution:
`enum(array('small', 'medium', 'large'))`
- `emptyValue()`
if your list value is not mandatory, it will allow user to select an empty value inside a drop-down. This method allows you to specify the value which will be presented to the user on that option: `emptyValue('Select Size')`;

Low-level properties

There are some properties which are used on a low-level and allows to change the way how field queries are created. Do not call those methods directly unless you know what you are doing:

- `from()`
specify joined table to which field belongs. Instead you can call `addField()` method on a join object.
- `actual()`
used to change the actual name of the field inside a table. Using second argument to `addField()` allows you to specify this value.

Specifying additional attributes

You may specify additional attributes by calling `setterGetter()` method:

```
$model->addField('bio')->setterGetter('hint', 'Type your bio here');
```

You would need to extend views to make them capable of interpreting your additional parameters. You can also access properties directly if you prefer it that way.

Creating Your Own Field Class

In several situations, you would want to have your own field class, such as:

- You are willing to change many properties to a different default at the same time.

- You would like to use custom behavior for query generations
- You wish to place hooks inside the models

If you are willing to create your own field, be sure to extend it from the "Field" class.

Expressions

Normally, when Model builds a query it asks every field listed in the "actual fields" list to add itself into `dsql()`. Fields will call the `field()` method to update the query of the model, from inside of the `updateSelectQuery()`. The "Field_Expression" redefines the method to insert user-defined expression into the model query.

Additionally expression field will also change number of flags. `editable()` will return false.

Expression field introduces one new property "expr" which stores your expression which can be expressed either as a string, as a DSQL expression or as a call-back. Use `set()` to specify the expression. Model have a method `addExpression()`, which will create expression field for you:

```
$model->addExpression('full_name')
->set('concat(name, " ", surname)');
```

When you are building expressions, be mindful that the fields you are referenced are SQL fields and are not model fields.

```
# Relational Model Use (ORM)
```

Let's do a quick review. First, we have created and abstracted SQL queries through a query builder. Next we have created and abstracted model fields. Now we need to tie them together through our ORM implementation and this will give us table abstraction. Let's create model for "Book"

```
class Model_Book extends Model_Table {
    public $table='book';
    function init(){
        parent::init();
        $this->addField('title');
        $this->addField('is_published')->type('boolean');
        $this->addField('cost')->type('money');
    }
}
```

Manipulating Model Fields

```
$m=$this->add('Model_Book');

$m['title']='Jungle Book';
$m['year']=123;

var_dump($m->get()); // shows title and year
```

The important thing about Agile Toolkit models, is that you can add more fields dynamically at any time.

```
$m->addField('abstract');
$m['abstract']='Lorem Ipsum ..';
```

Ability to take an existing model and add more fields allows us to extend existing models into new ones:

```
class Model_Published_Book extends Model_Book {
    function init(){
        parent::init();
        $this->addField('issn');
        $this->addCondition('is_published',true);
    }
}
```

but not only we can add additional fields, we can also add conditions, which would permanently change model's data-set.

Loading and Saving Models

```
$m1=$this->add('Model_Book');
$m1['title']='Jungle Book';
$m1->save();
echo $m1->id;    // will output book's id
echo $m1['is_published'];    // null

$m2=$this->add('Model_Published_Book');
$m2->tryLoad($m1->id);
echo $m2->loaded();    // false. Condition not met
```

Let's try this other way around:

```
$m1=$this->add('Model_Book_Published');
$m1['title']='Jungle Book';
$m1->save();
echo $m1->id;    // will output book's id
echo $m1['is_published'];    // true

$m2=$this->add('Model_Book');
$m2->tryLoad($m1->id);
echo $m2->loaded();    // true
```

Models can be loaded by using either ID key or other field:

```
$m=$this->add('Model_User');
$m->load(1);    // loads by id

$m->loadBy('email',$email);

$m->loadBy('id',$array_of_ids);

$m->orderBy('title')->tryLoadAny(); // loads first record
```

So far no surprises. Any model can also produce a DSQL of itself to be used to build a custom queries:

```
$m=$this->add('Model_User');
$m->count()->getOne();    // returns number

$m->sum('age')->getOne();    // returns age

$m->dsql()->del('fields')->field('avg(age)')->getOne();
    // custom field query
```

One of the up-sides of Agile Toolkit ORM is it's support for expressions. Let's go back to our Book / Chapter example:

```
$q=$this->add('Model_Book')->dsql()->del('fields')
    ->field('id')->where('is_published',1);

$c=$this->add('Model_Chapter')
    ->addCondition('book_id',$q);
```

Now the model \$c will have it's data-set dynamically restricted to only published books. Let's create some data:

```

$m1=$this->add('Model_Book')->set('name','Jungle Book')
->set('is_published',true)->saveAndUnload();

$m2=$this->add('Model_Book')->set('name','jQuery Book')
->set('is_published',false)->save();

$c->set('name','Jungle Chapter 1')->set('book_id',$m1->id)
->save(); // will be successful

$c->set('name','jQuery Chapter 1')->set('book_id',$m2->id)
->save(); // will fail

```

What about browsing:

```

echo $c->count()->debug()->getOne();

// select count(*) from chapter where book_id in
// (select id from book where is_published=1)

```

We can use this technique again for the Section model, however this time, we will use a method `fieldQuery()`:

```

$s=$this->add('Model_Section')
->addCondition('chapter_id',$c->fieldQuery('id'));

echo $s->count()->getOne();

// select count(*) from section where chapter_id in
// (select id from chapter where book_id in
// (select id from book where is_published=1))

```

This is cool, but too much typing and manipulating with the models. I am going to show you a small example from the further chapters on how this can be simplified:

```

echo $this->add('Model_Book')
->addCondition('is_published',true)
->ref('Chapter/Section')
->count()->getOne();

// select count(*) from section where chapter_id in
// (select id from chapter where book_id in
// (select id from book where is_published=1))
// - although may also use "join" if appropriate

```

As you can see in the examples, you can achieve things on a low level with some effort, but the low level gets abstracted more and more to reveal new beautiful syntax.

Model Aliasing

In some situations you must resort to table aliases to avoid clashes:

```

// select a.id,a.title,(select count(*) from book b
// where b.author_id=a.author_id) cnt from book a

```

The query above shows number of books written by the same author. But how do you write this code with the ORM? First let's try to build the query for our expression:

```

$q=$this->add('Model_Book',array('alias'=>'b'))->count();
$m=$this->add('Model_Book',array('alias'=>'a'));
$m->addExpression('cnt')->set($q
->where($q->getField('author_id'),
    $m->getElement('author_id')
);

```

This might seem confusing, let's clean it up:

```
$m=$this->add('Model_Book',array('alias'=>'a'));
$m->addExpression('cnt')->set(
    $m->newInstance(array('alias'=>'b'))
    ->addCondition('author_id',$m->getElement('author_id'))
    ->count()
);
```

Let's review what's happening there. First we create a model, but we pass alias property which will affect all the queries generated by that model to use the specific alias. Next we create a new instance of the book but this time we use a different alias. At this point we have 2 models, identical, but with different aliases.

Next I'm setting condition on a second model (b) where author_id field equals to author_id of the first model. It's done by passing Field object into a condition, which will then end up inside a DSQL object and will be properly expanded.

Finally our (b) model produces query for a count which is then used as an expression for the 'cnt' field.

Agile Toolkit will properly prefix fields with the table name if more than one table is used in a query.

Field Expressions

Agile Toolkit treats all types of fields with an absolute consistency. In other words, anywhere where a physical field can be used, you can also use expression. Here is an example:

```
$m=$this->add('Model_Author');
$m->addExpression('book_cnt')
    ->set($m->refSQL('Book')->count());

$m->addCondition('age',$m->getElement('book_cnt'));
```

This will create a model with only authors who wrote same amount of books as their age is. Someone who is 32 years old and wrote 32 books would match the condition.

The important point here is that 'age' could be either a physical field or another expression and regardless of that the model will properly build the query.

As a developer you have a full abstraction of a field and expressions.

Here is another example, which uses one field expression to build another field expression:

```
$m=$this->add('Model_Author');
$m->addExpression('full_name')
    ->set($m->dsql()->concat(
        $m->getElement('name'),' ',
        $m->getElement('surname')
    ));

$m->addExpression('full_email')->set(
    $m->dsql()->concat(
        $m->getElement('email'),' <',
        $m->getElement('name'),'>'
    ));
```

You will begin to appreciate these qualities as your project grows. Agile Toolkit is not only trying to help build query. It provides PROPER abstraction of query fields and expressions.

Any of the models above can be used with views or for addressing individual records. There are other ways to use expressions and I'll get back to them in further sections.

Iterating through records

When model is being iterated, it produces the id / data array pair.

```
foreach($this->add('Book') as $id=>$book){
    echo "id: ".$book['title']."\n";
}
// 1: Jungle Book
// 2: jQuery Book
```

The actual model will be also loaded and can be used much more flexibly than the produced array. Also you must note that Agile Toolkit will never retrieve a full result set unless you specifically ask for it. The iterations will happen as you read data. Here is a slightly different format:

```
foreach($book=$this->add('Book') as $junk){
    echo $book->id.": ".$book['title']."\n";
}
// 1: Jungle Book
// 2: jQuery Book
```

Be mindful of the expressions here. Book object is created first, then it's assigned to \$book variable. Then the object is iterated and stores results in \$junk array. We are not really interested in it, but we work with object directly.

One way is to update records on the fly here or perform traversal:

```
foreach($book=$this->add('Book') as $junk){
    echo $book->id.": ".$book['title']." by ".
        $book->ref('Author')->get('name')."\n";
}
// 1: Jungle Book by Joe Blogs
// 2: jQuery Book by John Smith
```

WARNING: traversing references inside a loop is very bad for scalability. Your previous favorite ORM might left you no other options, but with Agile Toolkit you can:

```
$book=$this->add('Book');
$book->join('author')->addField('author_name','name');
foreach($book as $junk){
    echo $book->id.": ".$book['title']." by ".
        $book['author_name']."\n";
}
// 1: Jungle Book by Joe Blogs
// 2: jQuery Book by John Smith
```

You already saw how to create subqueries, but this example adds a join and pulls field from a join. Note that we are also adding field under a different alias.

Timestamps

Things like timestamps, soft-delete and other features tend to appear as a built-in feature in other ORMs. Agile Toolkit has no built-in support for either, but with the basic features of Agile Toolkit Models you can implement these features easily.

```
$model->addField('deleted')->system(true)->type('boolean');
$model->addCondition('deleted',false);
$model->addHook('beforeDelete',function($m){
    $m->set('deleted',true)->saveAndUnload();
    $m->breakHook(false);
});
```

Timestamps are equally easy to implement:

```
$model->addField('created')->type('datetime')
    ->defaultValue($model->dsql()->expr('now()'));
$model->addField('updated')->type('datetime');
$model->addHook('beforeSave',function($m){
    $m['updated']=$m->dsql()->expr('now()');
});
```

How about if we pack this functionality into a nice controller?

```
class Controller_SoftDelete extends AbstractController {
    function init(){
        parent::init();
        $this->owner->addField('deleted')
            ->system(true)->type('boolean');
        $this->owner->addCondition('deleted',false);
        $this->owner->addHook('beforeDelete',function($m){
            $m->set('deleted',true)->saveAndUnload();
            $m->breakHook(false);
        });
    }
}

$model->setController('SoftDelete');    // simple use
```

Relations

You may define relations between models like this:

```
$book->hasMany('Chapters');
$book->hasOne('Author');
```

Later you can traverse by using `ref()` or `refSQL()`. For example, let's create a CRUD for editing list of chapters of a current book:

```
$this->add('CRUD')->setModel(
    $this->add('Model_Book')->load($id)->ref('Chapters')
);
```

You might have already noticed one example with refSQL. When you are defining relations you can also specify which fields to use. The next chapter will look into Model() in details.

This should just give you a brief idea about relations and I will detail them further.

Views

Agile Toolkit is a PHP UI Framework and it contains an infrastructure for object-based user interface building similar to those you would find in desktop and mobile applications.

```
$config['greeting']='Hello from Agile Toolkit';
```