
Agile UI

Release

Jun 26, 2017

1	Overview of Agile UI	3
1.1	Agile UI Design Goals	4
1.1.1	1. Out of the box experience	4
1.1.2	2. Compact and easy to integrate	4
1.1.3	3. Compatible with RestAPI	4
1.1.4	4. Deploy and Scale	4
1.1.5	5. High-level Solution	4
1.1.5.1	Overview Example	5
1.1.6	Best use of Agile UI	6
1.2	Component Overview	6
1.2.1	Using Components	6
1.2.2	Factory	7
1.2.3	Templates	8
1.2.4	Layouts	8
1.3	Advanced techniques	9
1.3.1	Non-PHP dependencies	9
1.3.2	Events and Actions	9
1.3.3	Callbacks	9
1.3.4	Virtual Pages	9
1.3.5	Extending with Add-ons	10
1.4	Using Agile UI	10
1.4.1	Learning Agile Toolkit	10
1.4.2	Application Tutorials	11
1.4.3	Education	11
1.4.4	Commercial Project Strategy	11
1.5	Things Agile UI simplifies	11
1.5.1	Database abstraction	11
1.5.2	Cloud deployment	11
2	Quickstart	13
2.1	Requirements	13
2.2	Installing	13
2.3	Coding “Hello, World”	13
2.4	Data Persistence	14
2.5	Data Model	14
2.6	Form and CRUD Components	15

2.7	Grid and CRUD	15
2.8	Conclusion	16
2.9	More Tutorials	16
3	Core Concepts	17
3.1	App	17
3.1.1	App	17
3.1.1.1	Purpose of App class	17
3.1.1.2	Utilities by App	21
3.1.1.3	Application and Layout	22
3.2	Seed	24
3.2.1	Seed	24
3.2.1.1	Empty Seed	24
3.2.1.2	Dependency Injection	25
3.2.1.3	Additional cases	25
3.3	Render Tree	25
3.3.1	Render Tree	26
3.3.1.1	Initialization	26
3.3.1.2	Late initialization	26
3.3.1.3	Rendering outside	27
3.4	Templates	28
3.4.1	Template	28
3.4.1.1	Example Template	28
3.4.1.2	Detailed Template Manipulation	29
3.4.1.3	Using Template Engine directly	31
3.4.1.4	Views and Templates	35
3.4.1.5	Best Practices	36
3.4.1.6	Globally Recognized Tags	37
3.4.1.7	Internals of Template Engine	37
3.5	Agile Data Integration	38
3.5.1	Agile Data Integration	38
3.5.1.1	Static Data Arrays	38
3.5.1.2	Raw SQL Queries	39
4	Components	41
4.1	Views	41
4.1.1	Initializing Render Tree	42
4.1.2	Use of \$app property and Dependency Injection	43
4.1.3	Integration with Agile Data	43
4.1.4	UI Role and Classes	44
4.1.5	Special-purpose properties	45
4.1.6	Rendering of a Tree	45
4.1.7	Modifying rendering logic	46
4.1.8	Unique ID tag	46
4.1.9	Modifying Basic Elements	47
4.1.10	Rest of yet-to-document/implement methods and properties	47
4.2	Grid	48
4.2.1	Using Grid	48
4.2.2	Adding Menu Items	48
4.2.3	Adding Quick Search	49
4.2.4	Paginator	49
4.2.5	Actions	49
4.2.6	Selection	49
4.2.7	Sorting	49

4.2.8	Advanced Usage	50
4.3	Table	50
4.3.1	Using Table	50
4.3.1.1	Adding Additional Columns	50
4.3.1.2	Column Objects	51
4.3.1.3	Advanced Column Definitions	52
4.3.2	Table sorting	53
4.3.2.1	JavaScript Sorting	54
4.3.2.2	Injecting HTML	54
4.3.3	Table Data Handling	54
4.3.3.1	Table Rendering Steps	55
4.3.4	Dealing with Multiple formatters	55
4.3.4.1	Header and Footer	56
4.3.4.2	Redefining	56
4.3.5	Advanced Usage	56
4.3.5.1	Toolbar, Quick-search and Paginator	56
4.3.6	Column attributes and classes	56
4.3.6.1	Static Attributes and classes	57
4.3.6.2	Columns without fields	57
4.3.6.3	Using dynamic values	58
4.3.7	Standard Column Types	58
4.3.7.1	Link	58
4.3.7.2	Money	58
4.3.7.3	Status	58
4.3.7.4	Template	59
4.3.7.5	Checkbox	59
4.3.7.6	Actions	59
4.4	Forms	59
4.4.1	Creating Basic Forms	61
4.4.1.1	Adding Fields to a form	61
4.4.1.2	Integrating Form with a Model	62
4.4.2	Form Submit Handling	62
4.4.3	Form Layout	64
4.4.4	Semantic UI modifiers	65
4.5	Button	65
4.5.1	Icons	65
4.5.2	Linking	66
4.5.3	Complex Buttons	66
4.6	Paginator	66
4.6.1	Adding and Using	66
4.6.2	Range and Logic	67
4.6.3	Template	67
4.6.4	Dynamic Reloading	67
4.7	Miscellaneous Views	67
4.7.1	Icon	68
4.7.1.1	Special Icons	68
4.7.1.2	As a part of	68
4.7.2	Label	69
4.7.2.1	Basic Usage	69
4.7.3	HelloWorld	69
4.7.4	LoremIpsum	70
4.7.5	Columns	70
4.7.5.1	Rows	70
4.7.5.2	Responsiveness and Performance	71

5	JavaScript Mapping	73
5.1	Introduction	73
5.1.1	Actions	73
5.1.2	Events	74
5.1.3	Extending	75
5.1.4	Including JS/CSS	75
5.2	Building actions with jsExpressionable	75
5.2.1	JavaScript Chain Building	75
5.2.2	View to JS integration	76
5.3	jsExpression	77
5.3.1	Template of jsExpression	78
5.3.2	Writing JavaScript code	79
5.4	Reloading	79
6	Advanced Topics	81
6.1	Agile Data	81
6.2	Interface Stability	81
6.3	Testing and Enterprise Use	82
6.3.1	Unit Tests	82
6.3.2	Business Logic Unit Tests	82
6.3.3	Integration Database Tests	82
6.3.4	Component Tests	82
6.3.5	User Testing	83
7	Indices and tables	85
	PHP Namespace Index	87

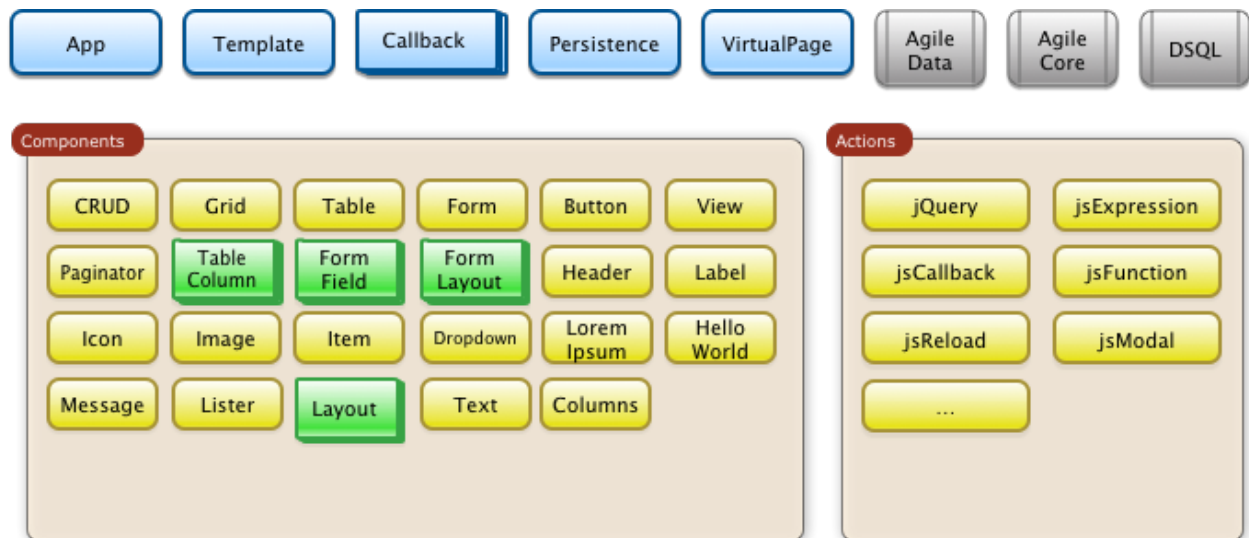
Contents:

Overview of Agile UI

Agile UI is a PHP component framework for building User Interfaces entirely in PHP. Although the components and Agile UI will typically on HTML, JavaScript, jQuery and CSS the goal of Agile UI is to abstract them away behind easy-to-use component object.

As a framework it's closely coupled with Agile Data (<http://agile-data.readthedocs.io>), which abstracts away database interaction operations. The default UI template set uses Semantic UI (<https://semantic-ui.com>) for presentation.

At a glance, Agile UI consists of the following:



Agile UI is designed and built for Agile Toolkit (<http://agiletoolkit.org/>) platform, with the goal to provide a user-friendly experience in creating data-heavy API / UI backend.

Agile UI Design Goals

Our goal is offer a free UI framework which you can use to develop even the most complex business application UI in just a few hours without diving deep into HTML/JS specifics.

1. Out of the box experience

Sample scenario:

If during .COM boom you purchased 1000 good-looking .COM domains and now selling them, you need to track offers from byers. You could use Excel but what if your staff need to access the data or you need to implement business operations such as accepting offers?

Agile UI is ideal for such scenario. By simply describing your data model, relations and operations you get a fully working UI and API with minimal setup.

2. Compact and easy to integrate

Simple scenario:

Your domains such as “happy.com” receives a lot of offers, so you want to place a special form for potential buyers to fill out. To weed out spammers you want to perform a address verification for filled-in data.

Agile UI contains a Form component which you can integrate into your existing app. More importantly it can securely access your offer database.

3. Compatible with RestAPI

Simple scenario:

You need a basic mobile app to check recent offers from your mobile phone.

You can set up API end-point for authorized access to your Offer database that follows same business rules and has access to same operations.

4. Deploy and Scale

Simple scenario:

You want to use serverless architecture where a 3rd party company is looking after your server, database and security, you simply provide your app.

Agile UI is designed and optimized for quick deployment into modern serverless architecture providers such as Heroku, Docker or even AWS Lambdas.

Agile UI / PHP application has a minimum “start-up” time, has the best CPU usage and gives you the highest efficiency and best scaling.

5. High-level Solution

Simple scenario:

You are a busy person who needs to get your application ready in 1h and then forget about it for the next few years. You are not particularly thrilled about digging through heaps of HTML, CSS or JS frameworks and need a solution which just works.

Overview Example

Agile UI / Agile Data code for your app can fit into a single file. See below for clarifications:

```
<?php
require 'vendor/autoload.php';

// Define your data structure
class Offer extends \atk4\data\Model {

    public $table = 'offer';

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

        // Persistence may not have structure, so we define here
        $this->addField('domain_name');
        $this->addFields(['contact_email', 'contact_phone']);
        $this->addField('date', ['type'=>'date']);
        $this->addField('offer', ['type'=>'money']);
        $this->addField('is_accepted', ['type'=>'boolean']);
    }
}

// Create Application object and initialize Admin Layout
$app = new \atk4\ui\App('Offer tracking system');
$app->initLayout('Admin');

// Connect to database and place a fully-interactive CRUD
$db = new \atk4\data\Persistence_SQL($dsn);
$app->layout->add(new \atk4\ui\CRUD())
    ->setModel(new Offer($db));
```

Through the course of this example, I am performing several core actions:

- *\$app* is an object representing your Web Application and abstracting all the input, output, error-handling and other technical implementation details of a standard web application.

In most applications you would want to extend this class yourself. When integrating Agile UI with MVC framework, you would be using a different App class, that properly integrates that framework capabilities.

For a *Component Overview* the App class provides level of abstraction and utility.

For full documentation see *App*.

- *\$db* this is database persistence object. It may be a Database which is either SQL or NoSQL but can also be RestAPI, a cache or a pseudo-persistence.

I have used Persistence_SQL class, which takes advantage of standard-compliant database server to speed up aggregation, multi-table and multi-record operations.

For a *Component Overview* the Persistence class provides data storage abstraction through the use of a Model class.

Agile Data has full documentation at <http://agile-data.readthedocs.io>.

- *Offer* is a Model - a database-agnostic declaration of your business entity. Model object represents a data-set for a specific persistence and conditions.

In our example, object is created representing all Offer records then passed into the CRUD *Component Overview*.

For a *Component Overview*, the Model represents information about the structure and offers mechanism to retrieve, store and delete data from *\$db* persistence.

- *CRUD* is a *Component Overview* class. Particularly *CRUD* is bundled with Agile UI and implements out-of-the-box interface for displaying data in a table format with operations to add, delete or edit record.

Although it's not obvious from the code, *CRUD* relies on multiple other components such as *Grid*, *Form*, *Menu*, *Paginator*, *Button*.

To sum this up in a more technical terms, Agile UI:

- Full abstraction of Web technologies through components.
- Concise syntax to define UI layouts in PHP.
- Built-in security and safety.
- Decoupled from data storage/retrieval mechanism.
- Designed to be integrated into full-stack frameworks.
- Abstaining from duplicating field names, types or validation logic outside of Model class.

Best use of Agile UI

- Creating admin backend UI for data entry and dashboards in shortest time and with minimum amount of code.
- Building UI components which you are willing to use across multiple environments (Laravel, Wordpress, Drupal, etc)
- Creating MVP prototype for Web Apps.

Component Overview

The component is a fundamental building block of Agile UI. Each component is fully self-sufficient and creating a class instance is enough to make component work.

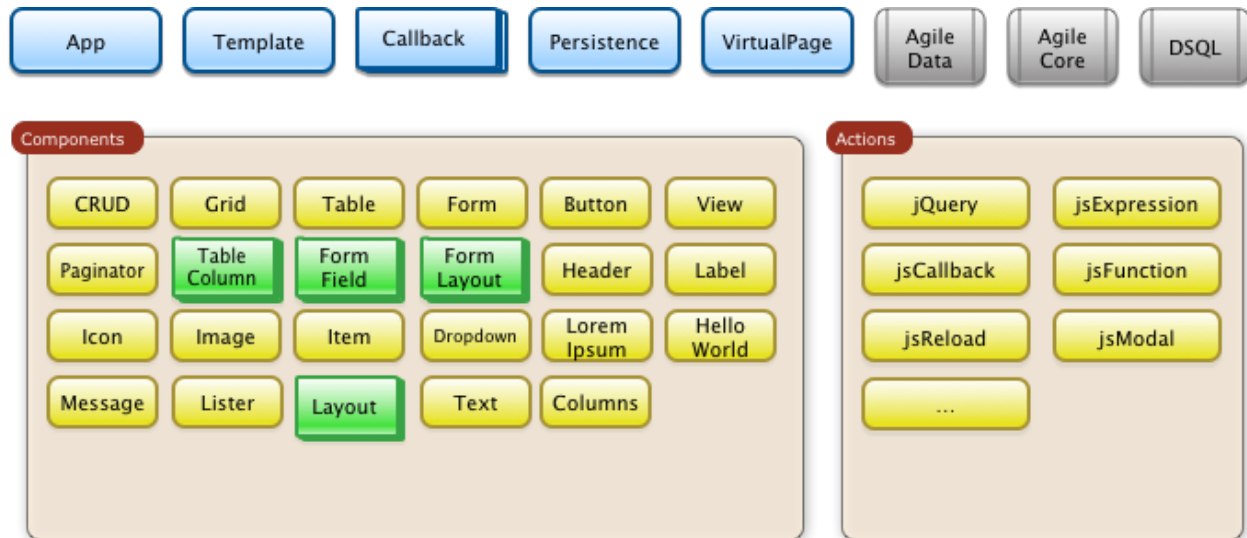
That means that components may rely on each-other and even though some may appear very basic to you, they are relied on by some other components for maximum flexibility. The next example adds “Cancel” button to a form:

```
$button = $form->add(new \atk4\ui\Button([
    'cancel',
    'icon'=>new \atk4\ui\Icon('pencil')
]))->link('dashboard.php');
```

Button and *Icon* are some of the most basic components in Agile UI. You will find *CRUD* / *Form* / *Grid* components much more useful:

Using Components

Look above at the *Overview Example*, component *GRID* was made part of application layout with a line:



```
$app->layout->add(new \atk4\ui\CRUD());
```

To render component individually and get the HTML and JavaScript use this format:

```
$form = new Form();
$form->init();
$form->setModel(new User($db));

$html = $form->render();
```

This would render an individual component and will return HTML / JavaScript:

```
<script>
  ..form submit callback setup..
</script>
<div class="ui form">
  <form id="atk_form">
    ... fields
    ... buttons
  </form>
</div>
```

For other use-cases please look into `View::render()`

Factory

Factory is a mechanism which allow you to use shorter syntax for creating objects. Agile UI goal is to be simple to use and readable, so taking advantage of loose types in PHP language allows us to use an alternative shorter syntax:

```
$form->add(['Button', 'Cancel', 'icon'=>'pencil'])
->link('dashboard.php');
```

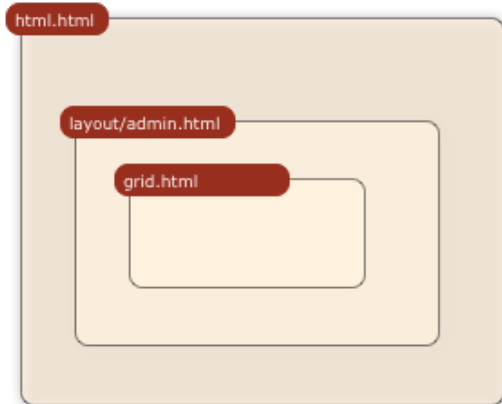
By default classes specified as 1st element of array passed to the `add()` method are resolved to namespace `atk4ui`, however the application class can fine-tune the search.

Usage of factory is optional. For more information see: <http://agile-core.readthedocs.io/en/develop/factory.html>

Templates

Components rely on *Template* class for parsing and rendering their HTML. The default template is written for Semantic UI framework, which makes sure that elements will look good and consistent.

Layouts

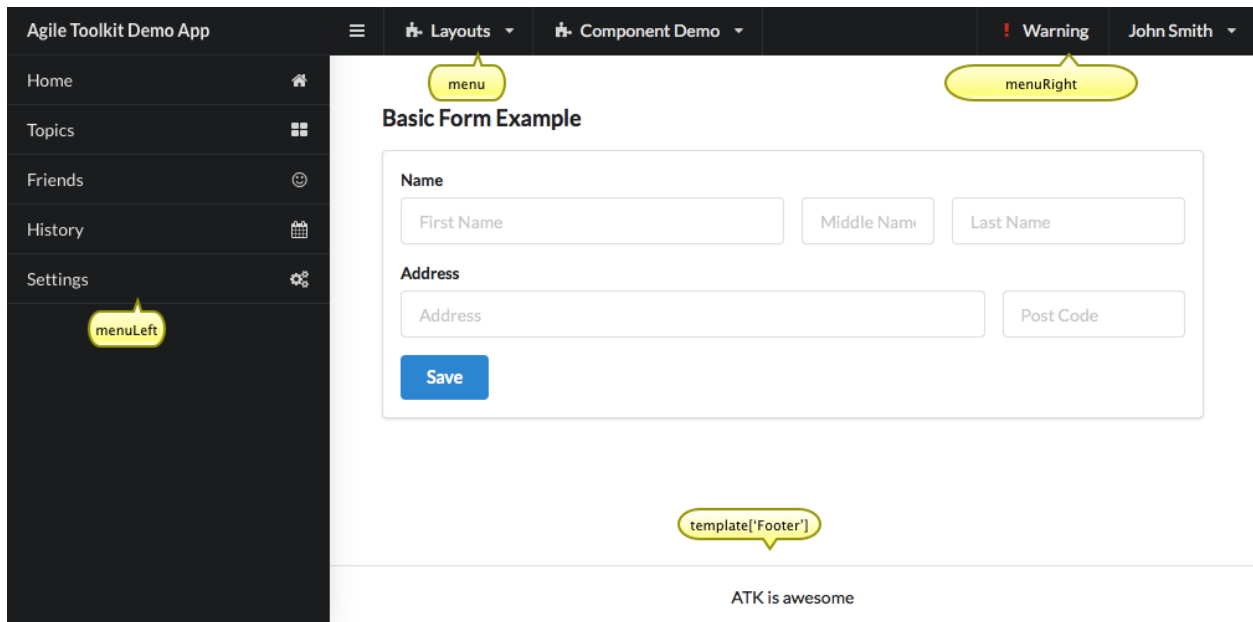


Using App class will utilise a minimum of 2 templates:

- html.html - boilerplate HTML code (<head>, <script>, <meta> and empty <body>)
- layout/admin.html - responsive layout containing page elements (menu, footer, etc)

As you add more components, they will appear inside your layout.

You'll also find that layout class such as `LayoutAdmin` is initializing some components on its own - sidebar menu, top menu.



If you are extending your Admin Layout, be sure to maintain same property names and then other components will make use of them, for example authentication controller will automatically populate a user-menu with the name of the user and log-out button.

Advanced techniques

By design we make sure that adding component into a Render Tree (See [Views](#)) is enough, so App provides a mechanism for components to:

- Depending on JS, CSS and other assets
- Define event handlers and actions
- Handle callbacks

Non-PHP dependencies

Your component may depend on additional JavaScript library, CSS or other files. At a present time you have to make them available through CDN and HTTPS. See: `App::requireJS`

Events and Actions

Agile UI allows you to initiate some JavaScript actions from inside PHP. The amount of application is quite narrow and is only intended for binding events between the components inside your component without involving developers who use your component in this process.

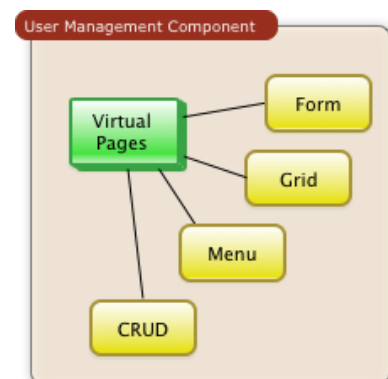
Callbacks

Some actions can be done only on the server side. For example adding a new record into the database.

Agile UI allows for component to do just that without no extra effort from you (such as setting up API routes). To make this possible, a component must be able to use unique URLs which will trigger the call-back.

To see how this is implemented, read about callback

Virtual Pages



Extending the concept of Callbacks, you can also define Virtual Pages. It is a dynamically generated URL which will respond with a partial render of your components.

Virtual Pages are useful for displaying UI on dynamic dialogs. As with everything else, virtual pages can be contained within the components so that no extra effort from you is required when component wishes to use dynamic modal dialog.

Extending with Add-ons

Agile UI is designed for data-agnostic UI components which you can add inside your application with a single line of code, but Agile Toolkit goes one way further by offering you a directory of published add-ons and install them by using a simple wizard.

Using Agile UI

Technologies advance forward to make it simpler and faster to build web apps. In some cases you can use ReactJS + Firebase but in most cases you will need to have a backend.

Agile Data is a most powerful framework for defining data-driven business model and Agile UI offers a very straightforward extension to attach your data to a wide range of standard UI widgets.

With this approach even the most complex business apps can be implemented in one day.

You can still implement ReactJS application by connecting it to the RestAPI endpoint provided by Agile Toolkit.

Warning: information on setting up API endpoints is coming soon.

Learning Agile Toolkit

We recommend that you start looking at Agile UI first. Continue reading through the *Quickstart* section and try building some basic apps. You will need to have a basic understanding of “code” and some familiarity with PHP language.

- QuickStart - 20-minute read and some code examples you can try.
- Core Concept - Read if you plan to design and build your own components.
 - Patterns and Principles
 - Views and common component properties/methods
 - Component Design and UI code refactoring
 - Injecting HTML Templates and Full-page Layouts
 - JavaScript Event Bindings and Actions
 - App class and Framework Integration
 - Usage Patterns
- Components - Reference for UI component classes
 - Button, Label, Header, Message, Menu, Column
 - Table and TableColumn
 - Form and Field
 - Grid and CRUD
 - Paginator
- Advanced Topics

If you are not interested in UI and only need Rest API, we recommend that you look into documentation for Agile Data (<http://agile-data.readthedocs.io>) and the Rest API extension (coming soon).

Application Tutorials

We have wrote few working cloud applications ourselves with Agile Toolkit and are offering you to look at their code. Some of them come with tutorials that teach you how to build application step-by-step.

Education

If you represent a group of students that wish to learn Agile Toolkit contact us about our education materials. We offer special support for those that want to learn how to develop Web Apps using Agile Toolkit.

Commercial Project Strategy

If you maintain a legacy PHP application and would like to have a free chat with us about some support and assistance, do not hesitate to reach out.

Things Agile UI simplifies

Some technologies are “prerequisites” in other PHP frameworks, but Agile Toolkit lets you develop a perfectly functional web application even if you are NOT familiar with technologies such as:

- HTML and Asset Management
- JavaScript, jQuery, NPM
- CSS styling, LESS
- Rest API and JSON

We do recommend that you come back and learn those technologies **after** you have mastered Agile Toolkit.

Database abstraction

Agile Data offers abstraction of database servers and will use appropriate query language to fetch your data. You may need to use SQL/NoSQL language of your database for some more advanced usage cases.

Cloud deployment

There are also ways to deploy your application into the cloud without knowledge of infrastructure, Linux and SSH. A good place to start is Heroku (<https://www.heroku.com/>). We reference Heroku in our tutorials, but Agile Toolkit can work with any cloud hosting that runs PHP apps.

In this section we will demonstrate how to build a very simple web application with just under 50 lines of PHP code. The important consideration here is that those are the **ONLY** lines you need to write. There are no additional code “generated” for you.

At this point you might not understand some concept, so I will provide referenced deeper into the documentation, but I suggest you to come back to this QuickStart to finish this simple tutorial.

Requirements

Agile Toolkit will work anywhere where PHP can. Find a suitable guide on how to set up PHP on your platform. Having a local database is a plus, but our initial application will work without persistent database.

Installing

Start your command-line and execute composer command:

```
composer require atk4/ui
```

Coding “Hello, World”

Open file *index.php* and add the following code:

```
<?php // 1
require 'vendor/autoload.php'; // 2

$app = new \atk4\ui\App('My First App'); // 3
$app->initLayout('Centered'); // 4
```

```
$app->layout->add('HelloWorld'); // 5
```

Clarifications

You should see the following output:



```
Hello World
```

Instead of manually outputting a text “Hello, World!” we have used a standard component. This actually brilliantly demonstrates a core purpose of Agile Toolkit. Instead of doing a lot of things yourself, you can rely on components that do things for you.

Data Persistence

To build our “ToDo” application, we need a good location to store list of tasks. We don’t really want to mess with the actual database and instead will use “SESSION” for storing data.

All components of Agile Data are database-agnostic and will not concern themselves with the way how you store data. I will start the session and connect `persistence` with it:

```
session_start();  
$s = new \atk4\data\Persistence_Array($_SESSION);
```

Data Model

We need a class `Task` which describes data model for the single ToDo item:

```
class ToDoItem extends \atk4\data\Model {  
    public $table = 'todo_item'; // 6  
    function init() {  
        parent::init();  
  
        $this->addField('name', ['caption'=>'Task Name', 'required'=>true]);  
        // 7  
  
        $this->addField('due', [  
            'type'=>'date', // 8  
            'caption'=>'Due Date',  
            'default'=>new \DateTime('+1 week') // 9  
        ]);  
    }  
};
```

```
}
}
```

Clarifications

As you might have noted already, Persistence and Model are defined independently from each-other.

Form and CRUD Components

Next we need to add Components that are capable of manipulating the data:

```
$col = $app->layout->add(['Columns', 'divided']); // 10
$col_reload = new \atk4\ui\jsReload($col); // 11

$form = $col->addColumn()->add('Form'); // 12
$form->setModel(new ToDoItem($s)); // 13
$form->onSubmit(function($form) use($col_reload) { // 14
    $form->model->save(); // 15

    return $col_reload; // 16
});

$col->addColumn() // 17
    ->add('Table')
    ->setModel(new ToDoItem($s));
```

Clarifications

It is time to test our application in action. Use the form to add new record data. Saving the form will cause table to also reload revealing new records.

Grid and CRUD

As mentioned before, UI Components in Agile Toolkit are often interchangeable, you can swap one for another. In our example replace right column (label 17) with the following code:

```
$grid = $col->addColumn()->add(['CRUD', 'paginator'=>false, 'ops'=>[ // 18
    'c'=>false, 'd'=>false // 19
]]);
$grid->setModel(new ToDo($s));

$grid->menu->addItem('Complete Selected', // 20
    new \atk4\ui\jsReload($grid->table, [ // 21
        'delete'=>$grid->addSelection()->jsChecked() // 22
    ])
);

if (isset($_GET['delete'])) { // 23
    foreach(explode(',', $_GET['delete']) as $id) { // 25
        $grid->model->delete($id);
    }
}
```

```
}  
}
```

Clarifications

Conclusion

We have just implemented a full-stack application with a stunning UI, advanced use of JavaScript, Form validation and reasonable defaults, calendar picker, multi-item selection in the grid with ability to also edit records through a dynamically loaded dialog.

All of that in about 50 lines of PHP code. More importantly, this code is portable, can be used anywhere and does not have any complex requirements. In fact, we could wrap it up into an individual Component that can be invoked with just one line of code:

```
$app->layout->add(new TodoManager())->setModel(new TodoItem());
```

Just like that you could be developing more components and re-using existing ones in your current or next web application.

More Tutorials

If you have enjoyed this tutorial, we have prepared another one for you, that builds a multi-page and multi-user application and takes advantage of database expressions, authentication and introduces more UI components:

- <https://github.com/atk4/money-lending-tutorial>
- (Demo: <https://money-lending-tutorial.herokuapp.com>)

Agile Toolkit and Agile UI is built by following the core concepts. Understanding the concepts is very important especially if you plan to write and distribute your own add-ons.

App

In any Agile UI application you would always need to have an App class. Even if you do not create this class explicitly, components generally will do it for you, but the common pattern is:

```
$app = new \atk4\ui\App('My App');  
$app->initLayout('Centered');  
$app->layout->add('LoremIpsum');
```

App

class App

App is a mandatory object that's essential for Agile UI to operate. If you don't create App object explicitly, it will be automatically created if you execute *\$component->init()* or *\$component->render()*.

In most use-scenarios, however, you would create instance of App class yourself before other components:

```
$app = new \atk4\ui\App('My App');  
$app->initLayout('Centered');  
$app->layout->add('LoremIpsum');
```

Purpose of App class

As you add one component into another, they will automatically inherit reference to App class. App class is ideal place to have all your environment configured and all the dependencies defined that other parts of your applications may require.

Most standard classes, however, will refrain from having too much assumptions about the App class, to keep overall code portable.

There may be some cases, when it's necessary to have multiple \$app objects, for example if you are executing unit-tests, you may want to create new App instance. If your application encounters exception, it will catch it and create a new App instance to display error message ensuring that error is not repeated.

Using App for Injecting Dependencies

Since App class becomes available for all objects and components of Agile Toolkit, you may add properties into the App class:

```
$app->db = new \atk4\ui\Persistence_SQL($dsn);  
  
// later anywhere in the code:  
  
$m = new MyModel($this->app->db);
```

Important: \$app->db is NOT a standard property. If you use this property, that's your own convention.

Using App for Injecting Behaviour

You may use App class hook to impact behaviour of your application:

- using hooks to globally impact object initialization
- override methods to create different behaviour, for example url() method may use advanced router logic to create beautiful URLs.
- you may re-define set-up of PersistenceUI and affect how data is loaded from UI.
- load templates from different files
- use a different CDN settings for static files

Using App as Initializer Object

App class may initialize some resources for you including user authentication and work with session. My next example defines property \$user and \$system for the app class to indicate a system which is currently active. (See system_pattern):

```
class Warehouse extends \atk4\ui\App  
{  
    public $user;  
    public $company;  
  
    function __construct($auth = true) {  
        parent::__construct('Warehouse App v0.4');  
  
        // My App class will establish database connection  
        $this->db = new \atk4\data\Persistence_SQL($_CLEARDB_DATABASE_URL['DSN']);  
        $this->db->app = $this;  
  
        // My App class provides access to currently logged user and currently_  
        ↪selected system.
```



```

$this->user = new User($this->db);
$this->company = new Company($this->db);
session_start();

// App class may be used for pages that do not require authentication
if (!$auth) {
    $this->initLayout('Centered');
    return;
}

// Load User from database based on session data
if (isset($_SESSION['user_id'])) {
    $this->user->tryLoad($_SESSION['user_id']);
}

// Make sure user is valid
if (!$this->user->loaded()) {
    $this->initLayout('Centered');
    $this->layout->add(['Message', 'Login Required', 'error']);
    $this->layout->add(['Button', 'Login', 'primary']->link('index.php'));
    exit;
}

// Load company data (System) for present user
$this->company = $this->user->ref('company_id');

$this->initLayout('Admin');

// Add more initialization here, such as populating menu.
}
}

```

After declaring your Application class like this, you can use it conveniently anywhere:

```

include 'vendor/autoload.php';
$app = new Warehouse();
$app->layout->add('CRUD')
    ->setModel($app->system->ref('Order'));

```

Quick Usage and Page pattern

A lot of the documentation for Agile UI uses a principle of initializing App object first, then manually add the UI elements using a procedural approach:

```

$app->layout->add('HelloWorld');

```

There is another approach in which your application will determine which Page class should be used for executing the request, subsequently creating setting it up and letting it populate UI (This behaviour is similar to Agile Toolkit prior to 4.3).

In Agile UI this pattern is implemented through a 3rd party add-on for page_manager and routing. See also `App::url()`

Clean-up and simplification

```
App::run()
```

```
property App::$run_called
```

```
property App::$is_rendering
```

```
property App::$always_run
```

App will also does certain actions to simplify handling of the application. For instance, App class will render itself automatically at the end of the application, so you can safely add objects into the *App* without actually triggering a global execution process:

```
$app->layout->add('HelloWorld');  
  
// Next line is optional  
$app->run();
```

If you do not want application to automatically execute *run()* you can either set *\$always_run* to false or use *terminate()* to the app with desired output.

Exception handling

```
App::caughtException()
```

```
property App::$catch_exception
```

By default App will also catch unhandled exceptions and will present them nicely to the user. If you have a better plan for exception, place your code inside try / catch block.

When Exception is caught, it's displayed using a 'Centered' layout and execution of original application is terminated.

Integration with other Frameworks

If you use Agile UI in conjunction with another framework, then you may be using a framework-specific App class, that implements tighter integration with the host application or full-stack framework.

```
App::requireJS()
```

Method to include additional JavaScript file in page:

```
$app->requireJS('https://code.jquery.com/jquery-3.1.1.js');  
$app->requireJS('https://cdnjs.cloudflare.com/ajax/libs/semantic-ui/2.2.10/semantic.  
↪min.js');
```

Using of CDN servers is always better than storing external libraries locally. Most of the time CDN servers are faster (cached) and more reliable.

```
App::requireCSS($url)
```

Method to include additional CSS stylesheet in page:

```
$app->requireCSS('//semantic-ui.com/dist/semantic.css');
```

```
App::initIncludes()
```

Initializes all includes required by Agile UI. You may extend this class to add more includes.

`App::getRequestURI()`

Decodes current request without any arguments. If you are changing URL generation pattern, you probably need to change this method to properly identify current page. See `App::url()`

Utilities by App

App provides various utilities that are used by other components.

`App::getTag()`

`App::encodeAttribute()`

`App::encodeHTML()`

Apart from basic utility, App class provides several mechanisms that are helpful for components.

Sticky GET Arguments

`App::stickyGet()`

`App::stickyForget()`

Problem: sometimes certain PHP code will only be executed when GET arguments are passed. For example, you may have a file `detail.php` which expects `order_id` parameter and would contain a *CRUD* component.

Since *CRUD* component is interactive, it may want to generate request to itself, but it must also include `order_id` otherwise the scope will be incomplete. Agile UI solves that with StickyGet arguments:

```
$order_id = $app->stickyGet('order_id');
$crud->setModel($order->load($order_id)->ref('Payment'));
```

This make sure that pagination, editing, addition or any other operation that *CRUD* implemetns will always address same model scope.

If you need to generate URL that respects stickyGet arguments, use `App::url()`.

Execution Termination

`App::terminate(output)`

Used when application flow needs to be terminated preemptively. For example when call-back is triggered and need to respond with some JSON.

You can also use this method to output debug data. Here is comparison to `var_dump`:

```
// var_dump($my_var); // does not stop execution, draws UI anyway
$this->app->terminate(var_export($my_var)); // stops execution.
```

Execution state

property `App::$is_rendering`

Will be true if application is currently rendering recursively through the Render Tree.

Links

App::url (*page*)

Method to generate links between pages. Specified with associative array:

```
$url = $app->url(['contact', 'from'=>'John Smith']);
```

this method must respond with a properly formatted url such as:

```
contact.php?from=John+Smith
```

If value with key 0 is specified ('contact') it will be used as the name of the page. By default url() will use page as "contact.php?.." however you can define different behaviour through page_manager.

The url() method will automatically append values of arguments mentioned to *stickyGet()*, but if you need URL to drop any sticky value, specify value explicitly as *false*.

Includes

App::requireJS (*\$url*)

Includes header into the <head> class that will load JavaScript file from a specified URL. This will be used by components that rely on external javascript libraries.

Hooks

Application implements HookTrait (<http://agile-core.readthedocs.io/en/develop/hook.html>) and the following hooks are available:

- beforeRender
- beforeOutput

Application and Layout

When writing application that uses Agile UI you can either select to use individual components or make them part of a bigger layout. If you use the component individually, then it will at some point initialize internal 'App' class that will assist with various tasks.

Having composition of multiple components will allow them to share the app object:

```
$grid = new \atk4\ui\Grid();
$grid->setModel($user);
$grid->addPaginator();           // initialize and populate paginator
$grid->addButton('Test');       // initialize and populate toolbar

echo $grid->render();
```

All of the objects created above - button, grid, toolbar and paginator will share same value for the 'app' property. This value is carried into new objects through AppScopeTrait (<http://agile-core.readthedocs.io/en/develop/appscope.html>).

Adding the App

You can create App object on your own then add elements into it:

```
$app = new App('My App');
$app->add($grid);

echo $grid->render();
```

This does not change the output, but you can use the 'App' class to your advantage as a "Property Bag" pattern to inject your configuration. You can even use a different "App" class altogether, which is how you can affect the default generation of links, reading of GET/POST data and more.

We are still not using the layout, however.

Adding the Layout

Layout can be initialized through the app like this:

```
$app->initLayout('Centered');
```

This will initialize two new views inside the app:

```
$app->html
$app->layout
```

The first view is a HTML boilerplate - containing HEAD / BODY tags but not the body contents. It is a standard html5 doctype template.

The layout will be selected based on your choice - 'Centered', 'Admin' etc. This will not only change the overall page outline, but will also introduce some additional views.

Going with the 'Admin' layout will populate some menu objects. Each layout may come with several views that you can populate:

```
$app->initLayout('Admin');

// Add item into menu
$app->layout->menu->addItem('User Admin', 'admin');
```

Integration with Legacy Apps

If you use Agile UI inside a legacy application, then you may already have layout and some patterns or limitations may be imposed on the app. Your first job would be to properly implement the "App" and either modification of your existing class or a new class.

Having a healthy "App" class will ensure that all of Agile UI components will perform properly.

3rd party Layouts

You should be able to find 3rd party Layout implementations that may even be coming with some custom templates and views. The concept of a "Theme" in Agile UI consists of offering the following 3 things:

- custom CSS build from Semantic UI

- custom Layout(s) along with documentation
- additional or tweaked Views

Unique layouts can be used to change the default look and as a stand-in replacement to some of standard layouts or as a new and entirely different layout.

Seed

Agile UI is developed to be easy to read and with simple and concise syntax. We make use of dynamic nature of PHP, therefore two syntax patterns are supported everywhere:

```
$app->layout->add(new \atk4\ui\Button('Hello'));  
  
and  
  
$app->layout->add(['Button', 'Hello']);
```

Method add() supports arguments in a various formats and we call that “Seed”. The same format can be used elsewhere, for example:

```
$button->icon = 'book';
```

We call this format ‘Seed’ this section will explain how and where it is used.

Seed

When creating a view, you have a chance to pass an argument to it. We have decided to refer to this special argument as “seed” because it has multiple purposes and the structure may differ depending on the element.

The most trivial case of seeding is:

```
$button = new Button('Hello');
```

Here button is seeded with a string and the button interprets it by setting a label. Other Views may interpret seed differently, Icon will convert seed into a class:

```
$icon = new Icon('book');
```

The seed designed to be intuitive for reading and remembering rather than attaching it to a specific technical property. Here are some more examples:

```
$app = new App('Hello World'); // name of the app
```

Empty Seed

Some views may not use a seed (yet), they will still accept an empty seed:

```
$app = new App(); // will use name = 'Untitled'  
$form = new Form(); // no name yet
```

Dependency Injection

Seed is a great way for you to perform dependency injection because seed argument may be an array. If seed is specified as “array”, then the value with index “0” will have identical effect as not using array:

```
$button = new Button('Hello');

// same as

$button = new Button(['Hello']);
```

Once the zero-indexed value is located and extracted from the seed, the rest of the array will be used as a dependency-injection or “defaults”:

```
$button = new Button(['Learn', 'icon'=>new Icon('book')]);
```

This will set the “icon” property of a Button class to the specified value (object). Setting of an object properties in only possible, if the property is declared. Attempt to set non-existent property will result in exception:

```
$button = new Button(['Learn', 'my_property'=>123]);
```

Additional cases

An individual object may add more ways to deal with seed. For example, when dealing with button you can specify both the label and the class through the seed:

```
$button = new Button(['Learn', 'big teal', 'icon'=>new Icon('book')]);
```

The view will generally map non-existing property seeds into HTML class, although it is recommended to use `View::addClass` method:

```
$button = new Icon(['book', 'red'=>true]);

// same as

$button = new Icon('book');
$button->addClass('red');

// or because it's a button
$button = new Icon('red book');
```

Render Tree

Agile Toolkit allows you to create components hierarchically. What is really unique and awesome is that you can create and link multiple UI objects together before linking them with other chunks of your UI:

```
$msg = new \atk4\ui\Message('Hey There');
$msg->add(new \atk4\ui\Button('Button'));

$app->layout->add($msg);
```

To find out more about how components are linked up together and rendered, see:

Render Tree

Agile UI allows you to create and combine various objects into a single Render Tree for unified rendering. Tree represents all the UI components that will contribute to the HTML generation. Render tree is automatically created and maintained:

```
$view = new \atk4\ui\View();  
$view->add(new Button('test'));  
echo $view->render();
```

When render on the \$view is executed, it will render button first then incorporate HTML into it's own template before rendering.

Here is a breakdown of how the above code works:

1. new instance of View is created and assigned to \$view.
2. new instance of Button.
3. Button object is registered as a “pending child” of a view.

At this point Button is NOT element of a view just yet. This is because we can't be sure if \$view will be rendered individually or will become child of another view. Method init() is not executed on either objects.

4. render() method will call renderAll()
5. renderAll will find out that the \$app property of a view is not set and will initialize it with default App.
6. renderAll will also find out that the init() has not been called for the \$view and will call it.
7. init() will identify that there are some “pending children” and will add them in properly.

Most of the UI classes will allow you to operate even if they are not initialized. For instance calling 'setModel()' will simply set a \$model property and does not really need to rely on \$api etc.

Next, lets look at what Initialization really is and why is it important.

Initialization

Calling init() method of a view is essential before any meaningful work can be done with it. This is important, because the following actions are performed:

- template is loaded (or cloned from parent's template)
- \$app property is set
- \$short_name property is determined
- unique \$name is assigned.

Many of UI components rely on the above to function properly. For example Grid will look for certain regions in it's template to clone them into separate objects. This cloning can only take place inside init() method.

Late initialization

When you create an application and select a Layout, the layout is automatically initialized:


```
$app = new \atk4\ui\App();
$app->setLayout('Centered');

echo $app->layout->name; // present, because layout is initialized!
```

After that, adding any objects into layout will initialize those objects too:

```
$b = $app->layout->add(new Button('Test1'));

echo $b->name; // present, because button was added into initialized object.
```

If object cannot determine the path to the application, then it will remain uninitialized for some time. This is called “Late initialization”:

```
$v = new Buttons();
$b2 = $v->add(new Button('Test2'));

echo $b2->name; // not set!! Not part of render tree
```

At this point, if you execute `$v->render()` it will create it’s own App and will create it’s own render tree. On other hand if you add `$v` inside layout, trees will merge and the same `$app` will be used:

```
$app->layout->add($v);

echo $b2->name; // fully set now and unique.
```

Agile UI will attempt to always initialize objects as soon as possible, so that you can get the most meaningful stack traces should there be any problems with the initialization.

Rendering outside

It’s possible for some views to be rendered outside of the app. In the previous section I speculated that calling `$v->render()` will create it’s own tree independent from the main one.

Agile UI sometimes uses the following approach to render element on the outside:

1. Create new instance of `$sub_view`.
2. Set `$sub_view->id = false`;
3. Calls `$view->_add($sub_view)`;
4. executes `$sub_view->renderHTML()`

This returns a HTML that’s stripped of any ID values, still linked to the main application but will not become part of the render tree. This approach is useful when it’s necessary to manipulate HTML and inject it directly into the template for example when embedding UI elements into Grid Column.

Since Grid Column repeats the HTML many times, the ID values would be troublesome. Additionally, the render of a `$sub_view` will be automatically embedded into the column and having it appear anywhere else on the page would be troublesome.

It’s usually quite futile to try and extract JS chains from the `$sub_tree` because JS wouldn’t work anyways, so this method will only work with static components.

Templates

Agile UI components store their HTML inside **.html* template files. Those files are loaded and manipulated by a Template class.

To learn more on how to create a custom template or how to change global template behaviour see:

Template

Agile UI relies on a lightweight built-in template engine to manipulate templates. The design goals of a template engine are:

- Avoid any logic inside template
- Keep easy-to-understand templates
- Allow preserving template content as much as possible

Example Template

Assuming that you have the following template:

```
Hello, {mytag}world{/}
```

Tags

the usage of *{* denotes a “tag” inside your HTML, which must be followed by alpha-numeric identifier and a closing *}*. Tag needs to be closed with either *{/mytag}* or *{/}*.

The following code will initialize template inside a PHP code:

```
$t = new Template('Hello, {mytag}world{/}');
```

Once template is initialized you can *render()* it any-time to get string “Hello, world”. You can also change tag value:

```
$t->set('mytag', 'Agile UI');  
  
// or  
  
$t['mytag'] = 'Agile UI';  
  
echo $t->render(); // "Hello, Agile UI".
```

Tags may also be self-closing:

```
Hello, {$mytag}
```

is identical to:

```
Hello, {mytag}{/}
```

Regions

We call region a tag, that may contain other tags. Example:

```
Hello, {$name}

{Content}
User {$user} has sent you {$amount} dollars.
{/Content}
```

When this template is parsed, region ‘Content’ will contain tags \$user and \$amount. Although technically you can still use *set()* to change value of a tag even if it’s inside a region, we often use Region to delegate rendering to another View (more about this in section for Views).

There are some operations you can do with a region, such as:

```
$content = $main_template->cloneRegion('Content');

$main_template->del('Content');

$content->set(['user'=>'Joe', 'amount'=>100]);
$main_template->append('Content', $content->render());

$content->set(['user'=>'Billy', 'amount'=>50]);
$main_template->append('Content', $content->render());
```

Usage in Agile UI

In practice, however, you will rarely have to work with the template engine directly, but you would be able to use it through views:

```
$v = new View('my_template.html');
$v['name'] = 'Mr. Boss';

$listener = new Listener($v, 'Content');
$listener->setModel($userlist);

echo $v->render();
```

The code above will work like this:

1. View will load and parse template.
2. Using `$v['name']` will set value of the tag inside template directly.
3. Listener will clone region ‘Content’ from my_template.
4. Listener will associate itself with provided model.
5. When rendering is executed, listener will iterate through the data, appending value of the rendered region back to \$v. Finally the \$v will render itself and echo result.

Detailed Template Manipulation

As I have mentioned, most Views will handle template for you. You need to learn about template manipulations if you are designing custom view that needs to follow some advanced patterns.

class `Template`

Template Loading

Array containing a structural representation of the template. When you create new template object, you can pass template as an argument to a constructor:

```
Template::__construct($template_string)  
    Will parse template specified as an argument.
```

Alternatively, if you wish to load template from a file:

```
Template::load($file)  
    Read file and load contents as a template.
```

```
Template::loadTemplateFromString($string)  
    Same as using constructor.
```

If the template is already loaded, you can load another template from another source which will override the existing one.

Template Parsing

Note: Older documentation.....

Opening Tag — alphanumeric sequence of characters surrounded by { and } (example {elephant})

Closing tag — very similar to opening tag but surrounded by {/ and }. If name of the tag is omitted, then it closes a recently opened tag. (example {/elephant} or {/})

Empty tag — consists of tag immediately followed by closing tag (such as {elephant}{/})

Self-closing tag — another way to define empty tag. It works in exactly same way as empty tag. ({\$elephant})

Region — typically a multiple lines HTML and text between opening and closing tag which can contain a nested tags. Regions are typically named with CamelCase, while other tags are named using snake_case:

```
some text before  
{ElephantBlock}  
    Hello, {$name}.  
  
    by {sender}John Smith{/}  
  
{/ElephantBlock}  
some text after
```

In the example above, sender and name are nested tags.

Region cloning - a process when a region becomes a standalone template and all of it's nested tags are also preserved.

Top Tag - a tag representing a Region containing all of the template. Typically is called `_top`.

Manually template usage pattern

Template engine in Agile Toolkit can be used independently, without views if you require so. A typical workflow would be:

1. Load template using `GiTemplate::loadTemplate` or `GiTemplate::loadTemplateFromString`.
2. Set tag and region values with `GiTemplate::set`.
3. Render template with `GiTemplate::render`.

Template use together with Views

A UI Framework such as Agile Toolkit puts quite specific requirements on template system. In case with Agile Toolkit, the following pattern is used.

- Each object corresponds to one template.
- View inserted into another view is assigned a region inside parents template, called `spot`.
- Developer may decide to use a default template, clone region of parents template or use a region of a user-defined template.
- Each View is responsible for it's unique logic such as repeats, substitutions or conditions.

As example, I would like to look at how `Form` is rendered. The template of form contains a region called "FormLine" - it represents a label and a input.

When an input is added into a Form, it adopts a "FormLine" region. While the nested tags would be identical, the markup around them would be dependent on form layout.

This approach allows you affect the way how `Form_Field` is rendered without having to provide it with custom template, but simply relying on template of a Form.

Popular use patterns for template engines	How Agile Toolkit implements it
Repeat section of template	<code>Listner</code> will duplicate Region
Associate nested tags with models record	<code>View</code> with <code>setModel()</code> can do that
Various cases within templates based on condition	<code>cloneRegion</code> or <code>get</code> , then use <code>set()</code>
Custom handling certain tags or regions	<code>GiTemplate::eachTag</code> with a callback
Filters (to-upper, escape)	all tags are escaped automatically, but other filters are not supported (yet)
Template inclusion	Generally discouraged, but can be done with <code>eachTag()</code>

Using Template Engine directly

Although you might never need to use template engine, understanding how it's done is important to completely grasp Agile Toolkit underpinnings.

Loading template

`Template::loadTemplateFromString` (*string*)

Initialize current template from the supplied string

`Template::loadTemplate` (*filename*)

Locate (using `PathFinder`) and read template from file

`Template::reload` ()

Will attempt to re-load template from it's original source.

`Template::__clone()`

Will create duplicate of this template object.

property `Template::$template`

Array structure containing a parsed variant of your template.

property `Template::$tags`

Indexed list of tags and regions within the template for speedy access.

property `Template::$template_source`

Simply contains information about where the template have been loaded from.

property `Template::$original_filename`

Original template filename, if loaded from file

Template can be loaded from either file or string by using one of following commands:

```
$template = $this->add('GiTemplate');  
$template->loadTemplateFromString('Hello, {name}world{/}');
```

To load template from file:

```
$template->loadTemplate('mytemplate');
```

And place the following inside `template/mytemplate.html`:

```
Hello, {name}world{/}
```

`GiTemplate` will use `PathFinder` to locate template in one of the directories of resource `template`.

Changing template contents

`Template::set` (*tag, value*)

Escapes and inserts value inside a tag. If passed a hash, then each key is used as a tag, and corresponding value is inserted.

`Template::setHTML` (*tag, value*)

Identical but will not escape. Will also accept hash similar to `set()`

`Template::append` (*tag, value*)

Escape and add value to existing tag.

`Template::appendHTML` (*tag, value*)

Similar to `append`, but will not escape.

Example:

```
$template = $this->add('GiTemplate');  
$template->loadTemplateFromString('Hello, {name}world{/}');  
  
$template->set('name', 'John');  
$template->appendHTML('name', '&nbsp;<i class="icon-heart"></i>');  
  
echo $template->render();
```

Using ArrayAccess with Templates

You may use template object as array for simplified syntax:

```
$template['name'] = 'John';

if(isset($template['has_title'])) {
    unset($template['has_title']);
}
```

Rendering template

Template::render()

Converts template into one string by removing tag markers.

Ultimately we want to convert template into something useful. Rendering will return contents of the template without tags:

```
$result=$template->render();

$this->add('Text')->set($result);
// Will output "Hello, World"
```

Template cloning

When you have nested tags, you might want to extract some part of your template and render it separately. For example, you may have 2 tags SenderAddress and ReceiverAddress each containing nested tags such as “name”, “city”, “zip”. You can’t use set(‘name’) because it will affect both names for sender and receiver. Therefore you need to use cloning. Let’s assume you have the following template in `template/envelope.html`:

```
<div class="sender">
{Sender}
    {$name},
    Address: {$street}
             {$city} {$zip}
{/Sender}
</div>

<div class="recipient">
{Recipient}
    {$name},
    Address: {$street}
             {$city} {$zip}
{/Recipient}
</div>
```

You can use the following code to manipulate the template above:

```
$template = $this->add('GiTemplate');
$template->loadTemplate('envelope');           // templates/envelope.html

// Split into multiple objects for processing
$sender    = $template->cloneRegion('Sender');
$recipient = $template->cloneRegion('Recipient');
```

```
// Set data to each sub-template separately
$sender    ->set($sender_data);
$recipient ->set($recipient_data);

// render sub-templates, insert into master template
$template->set('Sender',    $sender    ->render());
$template->set('Recipient', $recipient->render());

// get final result
$result=$template->render();
```

Same thing using Agile Toolkit Views:

```
$envelope = $this->add('View',null,null, ['envelope']);

$sender    = $envelope->add('View', null, 'Sender',    'Sender');
$recipient = $envelope->add('View', null, 'Recipient', 'Recipient');

$sender    ->tempalte->set($sender_data);
$recipient ->tempalte->set($recipient_data);
```

We do not need to manually render anything in this scenario. Also the template of \$sender and \$recipient objects will be appropriately cloned from regions of \$envelope and then substituted back after render.

In this example I've used a basic *View* class, however I could have used my own View object with some more sophisticated presentation logic. The only affect on the example would be name of the class, the rest of presentation logic would be abstracted inside view's `render()` method.

Other opeations with tags

Template::del(*tag*)

Empties contents of tag within a template.

Template::isSet(*tag*)

Returns true if tag exists in a template.

Template::trySet(*name, value*)

Attempts to set a tag, if it exists within template

Template::tryDel(*name*)

Attempts to empty a tag. Does nothing if tag with name does not exist.

Repeating tags

Agile Toolkit template engine allows you to use same tag several times:

```
Roses are {color}red{/}
Violets are {color}blue{/}
```

If you execute `set('color','green')` then contents of both tags will be affected. Similarly if you call `append('color','-ish')` then the text will be appended to both tags.

You can also use `eachTag()` to iterate through those tags.

Template::eachTag()

Executues a call-back for each tag

The format of the callback is:

```
function processTag($contents, $tag) {
    return ucwords($contents);
}
```

If your callback function defines second argument, then it will receive “unique” tag name which can be used to access template directly. This makes sense if you want to add object into that region. You can’t insert object into SMLite template, however every view in the system will have it’s template pre-initialized for you

The following template will implement the `include` functionality for your template:

```
$template->eachTag('include', function($content, $tag) use($template) {
    $t = $template->newInstance();
    $t->loadTemplate($content);
    $template->set($tag, $t->render());
});
```

See also: templates and views

Views and Templates

Let’s look how templates work together with View objects.

Default template for a view

`Template::defaultTemplate()`
Specify default template for a view.

By default view object will execute `defaultTemplate()` method which returns name of the template. This function must return array with one or two elements. First element is the name of the template which will be passed to `loadTemplate()`. Second argument is optional and is name of the region, which will be cloned. This allows you to have multiple views load data from same template but use different region.

Function can also return a string, in which case view will attempt to clone region with such a name from parent’s template. This can be used by your “menu” implementation, which will clone parent’s template’s tag instead to hook into some specific template:

```
function defaultTemplate() {
    return [ 'greeting' ]; // uses templates/greeting.html
}
```

Redefining template for view during adding

When you are adding new object, you can specify a different template to use. This is passed as 4th argument to `add()` method and has the same format as return value of `defaultTemplate()` method. Using this approach you can use existing objects with your own templates. This allows you to change the look and feel of certain object for only one or some pages. If you frequently use view with a different template, it might be better to define a new View class and re-define `defaultTemplate()` method instead:

```
$this->add('MyObject', null, null, array('greeting'));
```

Accessing view's template

Template is available by the time `init()` is called and you can access it from inside the object or from outside through “template” property:

```
$grid=$this->add('Grid',null,null,array('grid_with_hint'));
$grid->template->trySet('my_hint','Changing value of a grid hint here!');
```

In this example we have instructed to use a different template for grid, which would contain a new tag “my_hint” somewhere. If you try to change existing tags, their output can be overwritten during rendering of the view.

How views render themselves

Agile Toolkit perform object initialization first. When all the objects are initialized global rendering takes place. Each object's `render()` method is executed in order. The job of each view is to create output based on it's template and then insert it into the region of owner's template. It's actually quite similar to our Sender/Recipient example above. Views, however, perform that automatically.

In order to know “where” in parent's template output should be placed, the 3rd argument to `add()` exists — “spot”. By default spot is “Content”, however changing that will result in output being placed elsewhere. Let's see how our previous example with addresses can be implemented using generic views.

```
$envelope=$this->add('View',null,null,array('envelope'));
// 3rd argument is output region, 4th is template location
$sender=$envelope->add('View',null,'Sender','Sender');
$receiver=$envelope->add('View',null,'Receiver','Receiver');

$sender->template->trySet($sender_data);
$receiver->template->trySet($receiver_data);
```

Using Views with Templates efficiently

For maximum efficiency you should consider using Views and Templates in combination to achieve the result. The example which was previously mentioned under `GiTemplate::eachTag`:

```
$view->template->eachTag('include',function($content,$tag)use($view){
    $view->add('View',null,$tag,[$content]);
});
```

Best Practices

Don't use Template Engine without views

It is strongly advised not to use templates directly unless you have no other choice. Views implement consistent and flexible layer on top of `GiTemplate` as well as integrate with many other components of Agile Toolkit. The only cases when direct use of `SMLite` is suggested is if you are not working with HTML or the output will not be rendered in a regular way (such as RSS feed generation or TMail)

Organize templates into directories

Typically templates directory will have subdirectories: “page”, “view”, “form” etc. Your custom template for one of the pages should be inside “page” directory, such as page/contact.html. If you are willing to have a generic layout which you will use by multiple pages, then instead of putting it into “page” directory, call it page_two_columns.html.

You can find similar structure inside atk4/templates/shared or in some other projects developed using Agile Toolkit.

Naming of tags

Tags use two type of naming - CamelCase and underscore_lowercase. Tags are case sensitive. The larger regions which are typically used for cloning or by adding new objects into it are named with CamelCase. Examples would be: “Menu”, “Content” and “Recipient”. The lowercase and underscore is used for short variables which would be inserted into template directly such as “name” or “zip”.

Globally Recognized Tags

Agile Toolkit View will automatically substitute several tags with the values. The tag `{$_id}` is automatically replaced with a unique name by a View.

There are more templates which are being substituted:

- `{page}logout{/}` - will be replaced with relative URL to the page
- `{public}images/logo.png{/}` - will replace with URL to a public asset
- `{css}css/file.css{/}` - will replace with URL link to a CSS file
- `{js}jquery.validator.js{/}` - will replace with URL to JavaScript file

Avoid using the next two tags, which are obsolete:

- `{$atk_path}` - will insert URL leading to atk4 public folder
- `{$base_path}` - will insert URL leading to public folder of the project

Application (API) has a function `:php:'App_Web::setTags'` which is called for every view in the system. It's used to resolve “template” and “page” tags, however you can add more interesting things here. For example if you miss ability to include other templates from Smarty, you can implement custom handling for `{include}` tag here.

Be considered that there are a lot of objects in Agile Toolkit and do not put any slow code in this function.

Internals of Template Engine

When template is loaded, it's represented in the memory as an array. Example Template:

```
Hello {subject}world{/}!!
```

Content of tags are parsed recursively and will contain further arrays. In addition to the template tree, tags are indexed and stored inside “tags” property.

GiTemplate converts the template into the following structure available under `“$template->template”`:

```
// template property:
array (
  0 => 'Hello ',
  'subject#1' => array (
```

```
    0 => 'world',
  ),
  1 => '!!!',
)
```

Property tags would contain:

```
array (
  'subject'=> array( &array ),
  'subject#1'=> array( &array )
)
```

As a result each tag will be stored under its actual name and the name with unique “#1” appended (in case there are multiple instances of same tag). This allows `$smlite->get()` to quickly retrieve contents of appropriate tag and it will also allow `render()` to reconstruct the output efficiently.

Agile Data Integration

Agile UI framework is focused on building User Interfaces, but quite often interface must present data values to the user or even receive data values from user’s input.

Agile UI uses various techniques to present data formats, so that as a developer you wouldn’t have to sweat the details:

```
$user = new User($db);
$user->load(1);

$view = $app->layout-add(['template'=>'Hello, {$name}, your balance is {$balance}']);
$view->setModel($user);
```

Next section will explain you how Agile UI interacts with the data layer and how it outputs or inputs user data.

Agile Data Integration

Agile UI relies on Agile Data library for flexible access to user defined data sources. The purpose of this integration is to relieve developer from manually creating data fetching and storing code.

Other benefits of relying on Agile Data models is ability to store meta information of the models themselves. Without Agile UI as hard dependency, Agile UI would have to re-implement all those features on its own resulting in much bigger code footprint.

There are no way to use Agile UI without Agile Data, however Agile Data is flexibly enough to work with your own data sources. The rest of this chapter will explain how you can map various data structures.

Static Data Arrays

Agile Data contains `Persistence_Array` (<http://agile-data.readthedocs.io/en/develop/design.html?highlight=array#domain-model-actions>) implementation that load and store data in a regular PHP arrays. For the “quick and easy” solution Agile UI Views provide a method `View::setSource` which will work-around complexities and give you a syntax:

```
$grid->setSource([
  1 => ['name'=>'John', 'surname'=>'Smith', 'age'=>10],
  2 => ['name'=>'Sarah', 'surname'=>'Kelly', 'age'=>20],
]);
```

Note: Dynamic views will not be able to identify that you are working with static data and some features may not work properly. There are no plans in Agile UI to improve ways of using “setSource”, instead you should learn more how to use Agile Data for expressing your native data source. Agile UI is not optimized for setSource so it’s performance will generally be slower too.

Raw SQL Queries

Writing raw SQL queries is source of many errors, both with a business logic and security. Agile Data provides great ways for abstracting your SQL queries, but if you have to use a raw query:

```
// not sure how TODO - write this section.
```

Note: The above way to using raw queries has a performance implications, because Agile UI is optimised to work with Agile Data.

Contents:

Views

Agile UI is a component framework, which follows a software patterns known as *Render Tree* and *Two pass HTML rendering*.

class View

A View is a most fundamental object that can take part in the Render tree. All of the other components descend from the *View* class.

View object is recursive. You can take one view and add another View inside of it:

```
$v = new \atk4\ui\View(['ui'=>'segment', 'inverted']);  
$v->add(new \atk4\ui\Button(['Orange', 'inverted orange']));
```

The above code will produce the following HTML block:

```
<div class="ui inverted segment">  
  <button class="ui inverted orange button">Orange</button>  
</div>
```

All of the views combined form a *Render Tree*. In order to get the HTML output from all the *Views* in *Render Tree* you need to execute `render()` for the top-most leaf:

```
echo $v->render();
```

Each of the views will automatically render all of the child views.

Initializing Render Tree

Views use a principle of `delayed init`, which allow you to manipulate View objects in any way you wish, before they will actualized.

`View::add($object, $region = 'Content')`
Add child view as a parent of the this view.

In addition to adding a child object, sets up it's template and associate it's output with the region in our template.

Will copy `$this->app` into `$object->app`.

If this object is initialized, will also initialize `$object`

Parameters

- `$object` –
- `$region` –

`View::init()`

View will automatically execute an `init()` method. This will happen as soon as values for properties `app`, `id` and `path` can be determined.

You should override `init` method for composite views, so that you can `add()` additional sub-views into it.

In the next example I'll be creating 3 views, but it at the time their `__constructor` is executed it will be impossible to determine each view's position inside render tree:

```
$middle = new \atk4\ui\View(['ui'=>'segment', 'red']);
$stop = new \atk4\ui\View(['ui'=>'segments']);
$bottom = new \atk4\ui\Button(['Hello World', 'orange']);

// not arranged into render-tree yet

$middle->add($bottom);
$stop->add($middle);

// Still not sure if finished adding

$app = new \atk4\ui\App('My App');
$app->setLayout($stop);

// Calls init() for all elements recursively.
```

Each View's `init()` method will be executed first before calling the same method for child elements. To make your execution more straightforward we recommend you to create App class first and then continue with Layout initialization:

```
$app = new \atk4\ui\App('My App');
$stop = $app->setLayout(new \atk4\ui\View(['ui'=>'segments']));

$middle = $stop->add(new \atk4\ui\View(['ui'=>'segment', 'red']));

$bottom = $middle->add(new \atk4\ui\Button(['Hello World', 'orange']));
```

Finally, if you prefer a more concise code, you can also use the following format:

```
$app = new \atk4\ui\App('My App');
$stop = $app->setLayout('View', ['ui'=>'segments']);
```



```
$middle = $top->add('View', ['ui'=>'segment', 'red']);
$bottom = $middle->add('Button', ['Hello World', 'orange']);
```

The rest of documentaiton will use thi sconsise code to keep things readable, however if you value type-hinting of your IDE, you can keep using “new” keyword. I must also mention that if you specify first argument to add() as a string it will be passed to *\$app->factory()*, which will be responsible of instantiating the actual object.

(TODO: link to App:Factory)

Use of \$app property and Dependency Injeciton

property View::\$app

Each View has a property \$app that is defined through *atk4coreAppScopeTrait*. View elements rely on persistence of the app class in order to perform Dependency Injection.

Consider the following example:

```
$app->debug = new Logger('log'); // Monolog
// next, somewhere in a render tree
$view->app->debug->log('Foo Bar');
```

Agile UI will automatically pass your \$app class to all the views.

Integration with Agile Data

View::setModel(\$m)

Associate current view with a domain model.

property View::\$model

Stores currently associated model until time of rendering.

If you have used Agile Data, you should be familiar with a concept of creating Models:

```
$db = new \atk4\data\Persistence_SQL::connect($dsn);
$client = new Client($db); // extends \atk4\data\Model();
```

Once you have a model, you can associate it with a View such as Form or Grid so that those Views would be able to interact with your persistence directly:

```
$form->setModel($client);
```

In most environments, however, your application will rely on a primary Database, which can be set through your \$app class:

```
$app->db = new \atk4\data\Persistence_SQL::connect($dsn);
// next, anywhere in a view
$client = new Client($this->app->db);
$form->setModel($client);
```

Or if you prefer a more consise code:

```
$app->db = new \atk4\data\Persistence_SQL::connect($dsn);  
  
// next, anywhere in a view  
$form->setModel('Client');
```

Again, this will use *Factory* feature of your application to let you determine how to properly initialize the class corresponding to string 'Client'.

UI Role and Classes

```
View::__construct($defaults = [])
```

Parameters

- **\$defaults** – set of default properties and classes.

property View::\$ui

Indicates a role of a view for CSS framework.

A constructor of a view often maps into a `<div>` tag that has a specific role in a CSS framework. According to the principles of Agile UI, we support a wide variety of roles. In some cases, a dedicated object will exist, for example a Button. In other cases, you can use a View and specify a UI role explicitly:

```
$view = $layout->add('View', ['ui'=>'segment']);
```

If you happen to pass more key/values to the constructor or as second argument to `add()` they will be treated as default values for the properties of that specific view:

```
$view = $layout->add('View', ['ui'=>'segment', 'id'=>'test-id']);
```

For a more IDE-friendly format, however, I recommend to use the following syntax:

```
$view = $layout->add('View', ['ui'=>'segment']);  
$view->id = 'test-id';
```

You must be aware of a difference here - passing array to constructor will override default property before call to `init()`. Most of the components have been designed to work consistently either way and delay all the property processing until the render stage, so it should be no difference which syntax you are using.

If you are don't specify key for the properties, they will be considered an extra class for a view:

```
$view = $layout->add('View', ['inverted', 'orange', 'ui'=>'segment']);  
$view->id = 'test-id';
```

You can either specify multiple classes one-by-one or as a single string "inverted orange".

property View::\$class

List of classes that will be added to the top-most element during render.

```
View::addClass($class)
```

Add CSS class to element. Previously added classes are not affected. Multiple CSS classes can also be added if passed as space separated string or array of class names.

Parameters

- **\$class** (*string/array*) – CSS class name or array of class names

Returns \$this

View::removeClass(\$remove_class)

Parameters

- **\$remove_class** – string|array one or multiple classes to be removed.

In addition to the UI / Role classes during the render, element will receive extra classes from the \$class property. To add extra class to existing object:

```
$button->addClass('blue large');
```

Classes on a view will appear in the following order: “ui blue large button”

Special-purpose properties

A view may define a special-purpose properties, that may modify how the view is rendered. For example, Button has a property ‘icon’, that is implemented by creating instance of atk4uiIcon() inside the button.

The same pattern can be used for other scenarios:

```
$button = $layout->add('Button', ['icon'=>'book']);
```

This code will have same effect as:

```
$button = $layout->add('Button');
$button->icon = 'book';
```

During the Render of a button, the following code will be executed:

```
$button->add('Icon', ['book']);
```

If you wish to use a different icon-set, you can change Factory’s route for ‘Icon’ to your own implementation OR you can pass icon as a view:

```
$button = $layout->add('Button', ['icon'=>new MyAwesomeIcon('book')]);
```

Rendering of a Tree

View::render()

Perform render of this View and all the child Views recursively returning a valid HTML string.

Any view has the ability to render itself. Once executed, render will perform the following:

- call renderView() of a current object.
- call recursiveRender() to recursively render sub-elements.
- returns <script> with on-dom-ready instructions along with rendering of a current view.

You must not override render() in your objects. If you are integrating Agile UI into your framework you shouldn’t even use render(), but instead use getHTML and getJS.

View::getHTML()

Returns HTML for this View as well as all the child views.

View::getJS()

Return array of JS chains that was assigned to current element or it’s children.

Modifying rendering logic

When you creating your own View, you most likely will want to change it's rendering mechanics. The most suitable location for that is inside `renderView` method.

View::renderView()

Perform necessary changes in the `$template` property according to the presentation logic of this view.

You should override this method when necessary and don't forget to execute `parent::renderView()`.

property View::\$template

Template of a current view. The default value is 'element.html', however various UI classes will override this to use a different template, such as 'button.html'.

Before executing `init()` the template will be resolved and an appropriate Template object will assigned to this property. If null, will clone owner's `$region`.

property View::\$region

Name of the region in the owner's template where this object will output itself. By default 'Content'.

Here is a best practice for using custom template:

```
class MyView extends View {
    public $template = 'custom.html';

    public $title = 'Default Title';

    function renderView() {
        parent::renderView();
        $this->template['title'] = $this->title;
    }
}
```

As soon as the view becomes part of a render-tree, the Template object will also be allocated. At this point it's also possible to override default template:

```
$layout->add(new MyView(), ['template'=>$layout->template->cloneRegion('MyRegion')]);
```

Or you can set `$template` into object inside your constructor, in which case it will be left as-is.

On other hand, if your 'template' property is null, then the process of adding View inside RenderTree will automatically clone region of a parent.

Lister is a class that has no default template, and therefore you can add it like this:

```
$profile = $layout->add('View', ['template'=>'myview.html']);
$profile->setModel($user);
$profile->add('Lister', 'Tags')->setModel($user->ref('Tags'));
```

In this set-up a template `myview.html` will be populated with fields from `$user` model. Next, a Lister is added inside Tags region which will use the contents of a given tag as a default template, which will be repeated according to the number of referenced 'Tags' for given users and re-inserted back into the 'Tags' region.

See also *Template*.

Unique ID tag

property View::\$region

ID to be used with the top-most element.

Agile UI will maintain unique ID for all the elements. The tag is set through 'id' property:

```
$b = new \atk4\ui\Button(['id'=>'my-button3']);
echo $b->render();
```

Outputs:

```
<div class="ui button" id="my-button3">Button</div>
```

If ID is not specified it will be set automatically. The top-most element of a Render Tree will use `id=atk` and all of the child elements will create a derived ID based on it's UI role.

```
atk:
  atk-button:
  atk-button2:
  atk-form:
    atk-form-name:
    atk-form-surname:
    atk-form-button:
```

If role is unspecified then 'view' will be used. The main benefit here is to have automatic allocation of all the IDs throughout the render-tree ensuring that those ID's are consistent between page requests.

It is also possible to set the "last" bit of the ID postfix. When Form fields are populated, the name of the field will be used instead of the role. This is done by setting 'name' property.

property View::\$name

Specify a name for the element. If container already has object with specified name, exception will be thrown.

View::getJSID()

Return a unique ID for a given element based on owner's ID and our name.

Example:

```
$layout = new \atk4\ui\Layout(['id'=>'foo'])
$butt = $layout->add('Button', ['name'=>'bar']);o
echo $butt->getJSID(); // foo_bar
```

Modifying Basic Elements

TODO: Move to Element.

Most of the basic elements will allow you to manipulate their content, HTML attributes or even add custom styles:

```
$view->setElement('A');
$view->addStyle('align', 'right');
$view->addAttr('href', '');
```

Rest of yet-to-document/implement methods and properties

property View::\$skin

protected

Just here temporarily, until App picks it up

property View::\$content

Set static contents of this view.

View::**setProperty**(\$properties)

Parameters

- \$properties –

View::**setProperty**(\$key, \$val)

Parameters

- \$key –
- \$val –

View::**initDefaultApp**()

For the absence of the application, we would add a very simple one

View::**set**(\$arg1 = [], \$arg2 = null)

Parameters

- \$arg1 –
- \$arg2 –

View::**recursiveRender**()

Grid

class atk4\ui\Grid

If you didn't read documentation on [Table](#) you should start with that. While table implements the actual data rendering, Grid component supplies various enhancements around it, such as paginator, quick-search, toolbar and others by relying on other components.

Using Grid

Here is a simple usage:

```
$layout->add('Grid')->setModel(new Country($db));
```

To make your grid look nicer, you might want to add some buttons and enable quicksearch:

```
$grid = $layout->add('Grid');
$grid->setModel(new Country($db));

$grid->addQuickSearch();
$grid->menu->addItem('Reload Grid', new \atk4\ui\jsReload($grid));
```

Adding Menu Items

Grid top-bar which contains QuickSearch is implemented using Semantic UI "ui menu". With that you can add additional items and use all features of a regular Menu:

```
$sub = $grid->menu->addMenu('Drop-down');
$sub->addItem('Test123');
```

For compatibility grid supports addition of the buttons to the menu, but there are several Semantic UI limitations that wouldn't allow to format buttons nicely:

```
$grid->addButton('Hello');
```

If you don't need menu, you can disable menu bar entirely:

```
$grid = $layout->add(['Grid', 'menu' => false]);
```

Adding Quick Search

After you have associated grid with a model using `View::setModel()` you can include quick-search component:

```
$grid->addQuickSearch(['name', 'surname']);
```

If you don't specify argument, then search will be done by a models title field. (<http://agile-data.readthedocs.io/en/develop/model.html#title-field>)

Paginator

Grid comes with a paginator already. You can disable it by setting `$paginator` property to false. You can use `$ipp` to specify different number of items per page:

```
$grid->ipp = 10;
```

Actions

`Table` supports use of `TableColumnActions`, which allows to display button for each row. Calling `addAction()` provides a useful short-cut for creating column-based actions.

Selection

Grid can have a checkbox column for you to select elements. It relies on `TableColumnCheckbox`, but will additionally place this column before any other column inside a grid. You can use `TableColumnCheckbox::jsChecked()` method to reference value of selected checkboxes inside any `Actions`:

```
$sel = $grid->addSelection();
$grid->menu->addItem('show selection')->on('click', new \atk4\ui\jsExpression(
    'alert("Selected: "+[])', [$sel->jsChecked()]
));
```

Sorting

When grid is associated with a model that supports order, it will automatically make itself sortable. You can override this behaviour by setting `$sortable` property to `true` or `false`.

Additionally you may set list of sortable fields to a sortable property if you wish that your grid would be sortable only for those columns.

See also `Table::$sortable`.

Advanced Usage

You can use a different component instead of default `Table` by injecting `$table` property.

Table

Table is the simplest way to output multiple records of structured data. Table only works along with the model, however you can use `View::setSource` to inject static data (although it is slower than simply using a model).
`no_data`

Using Table

The simplest way to create a table:

```
$table = $layout->add('Table');  
$table->setModel(new Order($db));
```

The table will be able to automatically determine all the fields defined in your “Order” model, map them to appropriate column types, implement type-casting and also connect your model with the appropriate data source (database) `$db`.

To change the order or explicitly specify which columns must appear, you can pass list of columns as a second argument to `setModel`:

```
$table = $layout->add('Table');  
$table->setModel(new Order($db), ['name', 'price', 'amount', 'status']);
```

Table will make use of “Only Fields” feature in Agile Data to adjust query for fetching only the necessary columns. See also `field_visibility`.

Adding Additional Columns

If you feel that you’d like to add several other columns to your table, you need to understand what type of columns they would be.

If your column is designed to carry a value of any type, then it’s much better to define it as a Model Field. A good example of this scenario is adding “total” column to list of your invoice lines that already contain “price” and “amount” values. Start by adding new Field in the model that is associated with your table:

```
$table = $layout->add('Table');  
$order = new Order($db);  
  
$order->addExpression('total', '[price]*[amount]')->type = 'money';  
  
$table->setModel($order, ['name', 'price', 'amount', 'total', 'status']);
```

The type of the Model Field determines the way how value is presented in the table. I’ve specified value to be ‘money’ which makes column align values to the right, format it with 2 decimal signs and possibly add a currency sign.

To learn about value formatting, read documentation on `ui_persistence`.

Table object does not contain any information about your fields (such as captions) but instead it will consult your Model for the necessary field information. If you are willing to define the type but also specify the caption, you can use code like this:

```
$table = $layout->add('Table');
$order = new Order($db);

$order->addExpression('total', [
    '[price]*[amount]',
    'type'=>'money',
    'caption'=>'Total Price'
]);

$table->setModel($order, ['name', 'price', 'amount', 'total', 'status']);
```

Column Objects

Table object relies on a separate class: `atk4uiTableColumnGeneric` to present most of the values. The goals of the column object is to format anything around the actual values. The type = ‘money’ will result in a custom formatting of the value, but will also require column to be right-aligned. To simplify this, type = ‘money’ will use a different column class - `TableColumnMoney`. There are several others, but first we need to look at the generic column and understand it’s base capabilities:

class `atk4\ui\TableColumnGeneric`

A class responsible for cell formatting. This class defines 3 main methods that is used by the Table when constructing HTML:

`atk4\ui\TableColumnGeneric::getHeaderCellHTML` (*atk4dataField \$f*)

Must respond with HTML for the header cell (`<th>`) and an appropriate caption. If necessary will include “sorting” icons or any other controls that go in the header of the table.

The output of this field will automatically encode any values (such as caption), shorten them if necessary and localize them.

`atk4\ui\TableColumnGeneric::getTotalsCellHTML` (*atk4dataField \$f, \$value*)

Provided with the field and the value, format the cell for the footer “totals” column. Table can rely on various strategies for calculating totals. See `Table::addTotals`.

`atk4\ui\TableColumnGeneric::getDataCellHTML` (*atk4dataField \$f*)

Provided with a field, this method will respond with HTML **template**. In order to keep performance of Web Application at the maximum, Table will execute `getDataCellHTML` for all the fields once. When iterating, a combined template will be used to display the values.

The template must not incorporate field values (simply because related model will not be loaded just yet), but instead should resort to tags and syntax compatible with `Template`.

A sample template could be:

```
<td><b>{ $name }</b></td>
```

Note that the “name” here must correspond with the field name inside the Model. You may use multiple field names to format the column:

```
<td><b>{$year}-{$month}-{$day}</b></td>
```

The above 3 methods define first argument as a field, however it's possible to define column without a physical field. This makes sense for situations when column contains multiple field values or if it doesn't contain any values at all.

Sometimes you do want to inject HTML instead of using row values:

```
atk4\ui\TableColumnGeneric::getHTMLTags($model, $field = null)
```

Return array of HTML tags that will be injected into the row template. See *Injecting HTML* for further example.

Advanced Column Denifitions

```
class atk4\ui\Table
```

Table defines a method *columnFactory*, which returns Column object which is to be used to display values of specific model Field.

```
atk4\ui\Table::columnFactory(atk4dataField $f)
```

If the value of the field can be displayed by *TableColumnGeneric* then Table will respond with object of this class. Since the default column does not contain any customization, then to save memory Table will re-use the same objects for all generic fields.

```
property atk4\ui\Table::$default_column
```

Protected property that will contain “generic” column that will be used to format all columns, unless a different column type is specified or the Field type will require a use of a different class (e.g. ‘money’). Value will be initialized after first call to *Table::addColumn*

```
property atk4\ui\Table::$columns
```

Contains array of defined columns.

```
atk4\ui\Table::addColumn([$name], TableColumnGeneric $column = null, atk4uiDataField = null)
```

Adds a new column to the table. This method has several few usages, here is the most basic one:

```
$table->setModel(new Order($db), ['name', 'price', 'total']);  
$table->addColumn(new \atk4\ui\TableColumn\Delete());
```

The above code will add a new extra column that will only contain ‘delete’ icon. When clicked it will automatically delete the record.

You have probably noticed, that I have omitted the name for this column. If name is not specified (null) then the Column object will receive “null” when the call to *TableColumnGeneric::getHeaderCellHTML*, *TableColumnGeneric::getTotalsCellHTML* and *TableColumnGeneric::getDataCellHTML* will be made. The *TableColumnGeneric* will not be able to cope with this situations, but many other column types are perfectly fine with this.

Some column classes will be able to take some information from a specified column, but will work just fine if column is not passed.

If you do specify a string as a \$name for addColumn, but no such field exist in the model, the method will rely on 3rd argument to create a new field for you. Here is example that calculates the “total” column value (as above) but using PHP math instead of doing it inside database:

```
$table = $layout->add('Table');  
$order = new Order($db);  
  
$table->setModel($order, ['name', 'price', 'amount', 'status']);
```

```
$table->addColumn('total', new \atk4\data\Field\Calculated(
    function($row) {
        return $row['price'] * $row['amount'];
    }));
```

If you execute this code, you'll notice that the "total" column is now displayed last. If you wish to position it before status, you can use the final format of addColumn():

```
$table = $layout->add('Table');
$order = new Order($db);

$table->setModel($order, ['name', 'price', 'amount']);
$table->addColumn('total', new \atk4\data\Field\Calculated(
    function($row) {
        return $row['price'] * $row['amount'];
    }));
$table->addColumn('status');
```

This way we don't populate the column through setModel() and instead populate it manually later through addColumn(). This will use an identical logic (see *Table::columnFactory*). For your convenience there is a way to add multiple columns efficiently.

addColumnns (\$names);

Here, names can be an array of strings (['status', 'price']) or contain array that will be passed as argument to the addColumn method (['total', \$field_def], ['delete', \$delete_column]);

As a final note in this section - you can re-use column objects multiple times:

```
$c_gap = new \atk4\ui\TableColumn\Template('<td> ... <td>');

$table->addColumn($c_gap);
$table->setModel(new Order($db), ['name', 'price', 'amount']);
$table->addColumn($c_gap);
$table->addColumnns(['total', 'status'])
$table->addColumn($c_gap);
```

This will result in 3 gap columns rendered to the left, middle and right of your Table.

Table sorting

property atk4\ui\Table::\$sortable

property atk4\ui\Table::\$sort_by

property atk4\ui\Table::\$sort_order

Table does not support an interactive sorting on it's own, (but *Grid* does), however you can designate columns to display headers as if table were sorted:

```
$table->sortable = true;
$table->sort_by = 'name';
$table->sort_order = 'ascending';
```

This will highlight the column "name" header and will also display a sorting indicator as per sort order.

JavaScript Sorting

You can make your table sortable through JavaScript inside your browser. This won't work well if your data is paginated, because only the current page will be sorted:

```
$table->app->includeJS('http://semantic-ui.com/javascript/library/tablesort.js');  
$table->js(true)->tablesort();
```

For more information see <https://github.com/kylefox/jquery-tablesort>

Injecting HTML

The tag will override model value. Here is example usage of `TableColumnGeneric::getHTMLTags`:

```
class ExpiredColumn extends \atk4\ui\TableColumn\Generic  
{  
    public function getDataCellHTML()  
    {  
        return '{$_expired}';  
    }  
  
    function getHTMLTags($model)  
    {  
        return ['_expired'=>  
            $model['date'] < new \DateTime() ?  
            '<td class="danger">EXPIRED</td>' :  
            '<td></td>'  
        ];  
    }  
}
```

Your column now can be added to any table:

```
$table->addColumn(new ExpiredColumn());
```

IMPORTANT: HTML injection will work unless `Table::use_html_tags` property is disabled (for performance).

Table Data Handling

Table is very similar to `Listner` in the way how it loads and displays data. To control which data Table will be displaying you need to properly specify the model and persistence. The following two examples will show you how to display list of “files” inside your Dropbox folder and how to display list of issues from your Github repository:

```
// Show contents of dropbox  
$dropbox = \atk4\dropbox\Persistence($db_config);  
$files = new \atk4\dropbox\Model\File($dropbox);  
  
$layout->add('Table')->setModel($files);  
  
// Show contents of dropbox  
$github = \atk4\github\Persistence_Issues($github_api_config);  
$issues = new \atk4\github\Model\Issue($github);  
  
$layout->add('Table')->setModel($issues);
```

This example demonstrates that by selecting a 3rd party persistence implementation, you can access virtually any API, Database or SQL resource and it will always take care of formatting for you as well as handle field types.

I must also note that by simply adding 'Delete' column (as in example above) will allow your app users to delete files from dropbox or issues from GitHub.

Table follows a “universal data design” principles established by Agile UI to make it compatible with all the different data persistences. (see `universal_data_access`)

For most applications, however, you would be probably using internally defined models that rely on data stored inside your own database. Either way, several principles apply to the way how Table works.

Table Rendering Steps

Once model is specified to the Table it will keep the object until render process will begin. Table columns can be defined anytime and will be stored in the `Table::columns` property. Columns without defined name will have a numeric index. It's also possible to define multiple columns per key in which case we call them “formatters”.

During the render process (see `View::renderView`) Table will perform the following actions:

1. Generate header row.
2. Generate template for data rows.
3. **Iterate through rows**
 - 3.1 Current row data is accessible through `$table->model` property.
 - 3.2 Update Totals if `Table::addTotals` was used.
 - 3.3 Insert row values into `Table::t_row`
 - 3.3.1 Template relies on `ui_persistence` for formatting values
 - 3.4 Collect HTML tags from 'getHTMLTags' hook.
 - 3.5 Collect `getHTMLTags()` from columns objects
 - 3.6 Inject HTML into `Table::t_row` template
 - 3.7 Render and append row template to Table Body (`{Body}`)
 - 3.8 Clear HTML tag values from template.
4. If no rows were displayed, then “empty message” will be shown (see `Table::t_empty`).
5. If `addTotals` was used, append totals row to table footer.

Dealing with Multiple formatters

You can add column several times like this:

```
$table->addColumn('salary', new \atk4\ui\TableColumn\Money());
$table->addColumn('salary', new \atk4\ui\TableColumn\Link(['page2']));
```

In this case system needs to format the output as a currency and subsequently format it as a link. Formatters are always applied in the same orders they are defined. Remember that `setModel()` will typically set a Generic formatter for all columns.

There are a few things to note:

1. calling `addColumn` multiple time will convert `Table::columns` value for that column into array containing all column objects
2. formatting is always applied in same order as defined - in example above Money first, Link after.
3. output of the 'Money' formatter is used into Link formatter as if it would be value of cell.

`TableColumnMoney::getDataCellTemplate` is called, which returns ONLY the HTML value, without the `<td>` cell itself. Subsequently `TableColumnLink::getDataCellTemplate` is called and the `'{$salary}'` tag from this link is replaced by output from Money column resulting in this template:

```
<a href="{c_name_link}">£ {s_salary}</a>
```

To calculate which tag should be used, a different approach is done. Attributes for `<td>` tag from Money are collected then merged with attributes of a Link class. The money column wishes to add class “right aligned single line” to the `<td>` tag but sometimes it may also use class “negative”. The way how it’s done is by defining `class="{f_name_money}"` as one of the TD properties.

The link does add any TD properties so the resulting “td” tag would be:

```
['class' => ['{f_name_money}']] ]  
  
// would produce <td class="{f_name_money}"> .. </td>
```

Combined with the field template generated above it provides us with a full cell template:

```
<td class="{f_name_money}"><a href="{c_name_link}">£ {s_salary}</a></td>
```

Which is concatenated with other table columns just before rendering starts. The actual template is formed by calling. This may be too much detail, so if you need to make a note on how template caching works then,

- values are encapsulated for named fields.
- values are concatenated by anonymous fields.
- `<td>` properties are stacked
- last formatter will convert array with td properties into an actual tag.

Header and Footer

When using with multiple formatters, the last formatter gets to render Header column. The footer (totals) uses the same approach for generating template, however a different methods are called from the columns: `getTotalsCellTemplate`

Redefining

If you are defining your own column, you may want to re-define `getDataCellTemplate`. The `getDataCellHTML` can be left as-is and will be handled correctly. If you have overridden `getDataCellHTML` only, then your column will still work OK provided that it’s used as a last formatter.

Advanced Usage

Table is a very flexible object and can be extended through various means. This chapter will focus on various requirements and will provide a way how to achieve that.

Toolbar, Quick-search and Paginator

See *Grid*

Column attributes and classes

By default Table will include ID for each row: `<tr data-id="123">`. The following code example demonstrates how various standard column types are relying on this property:

```
$table->on('click', 'td', new jsExpression(
    'document.location=page.php?id=[]',
    [(new jQuery())->closest('tr')->data('id')]
));
```

See also *JavaScript Mapping*.

Static Attributes and classes

```
class atk4\ui\TableColumnGeneric
addClass($class, $scope = 'body');
setAttr($attribute, $value, $scope = 'body');
```

The following code will make sure that contents of the column appear on a single line by adding class “single line” to all body cells:

```
$table->addColumn('name', (new \atk4\ui\TableColumn\Generic()->addClass('single line
↳')));
```

If you wish to add a class to ‘head’ or ‘foot’ or ‘all’ cells, you can pass 2nd argument to addClass:

```
$table->addColumn('name', (new \atk4\ui\TableColumn\Generic()->addClass('right aligned
↳', 'all')));
```

There are several ways to make your code more readable:

```
$table->addColumn('name', new \atk4\ui\TableColumn\Generic()
->addClass('right aligned', 'all');
```

Or if you wish to use factory, the syntax is:

```
$table->addColumn('name', 'Generic')
->addClass('right aligned', 'all');
```

For setting an attribute you can use setAttr() method:

```
$table->addColumn('name', 'Generic')
->setAttr('colspan', 2, 'all');
```

Setting a new value to the attribute will override previous value.

Please note that if you are redefining `TableColumnGeneric::getHeaderCellHTML`, `TableColumnGeneric::getTotalsCellHTML` or `TableColumnGeneric::getDataCellHTML` and you wish to preserve functionality of setting custom attributes and classes, you should generate your TD/TH tag through getTag method.

```
getTag($tag, $position, $value);
```

Will apply cell-based attributes or classes then use `App::getTag` to generate HTML tag and encode it’s content.

Columns without fields

You can add column to a table that does not link with field:

```
$cb = $table->addColumn('Checkbox');
```

Using dynamic values

Body attributes will be embedded into the template by the default `TableColumnGeneric::getDataCellHTML`, but if you specify attribute (or class) value as a tag, then it will be auto-filled with row value or injected HTML.

For further examples of and advanced usage, see implementation of `TableColumnStatus`.

Standard Column Types

In addition to `TableColumnGeneric`, Agile UI includes several column implementations.

Link

class `atk4\ui\TableColumnLink`

Put `<a href=`. link over the value of the cell. The page property can be specified to constructor. There are two usage patterns. With the first you can specify full URL as a string:

```
$table->addColumn('name', new \atk4\ui\TableColumn\Link('http://google.com/?q={$name}'->));
```

The name value will be automatically inserted. The other option is to use page array:

```
$table->addColumn('name', new \atk4\ui\TableColumn\Link(['details', 'id'=> '{$id}', 'status'=> '{$status}']));
```

Money

class `atk4\ui\TableColumnMoney`

Helps formatting monetary values. Will align value to the right and if value is less than zero will also use red text. The money cells are not wrapped.

For the actual number formatting, see `ui_persistence`

Status

class `atk4\ui\TableColumnStatus`

Allow you to set highlight class and icon based on column value. This is most suitable for columns that contain pre-defined values.

If your column “status” can be one of the following “pending”, “declined”, “archived” and “paid” and you would like to use different icons and colors to emphasise status:

```
$states = [ 'positive'=>['paid', 'archived'], 'negative'=>['declined'] ];  
$table->addColumn('status', new \atk4\ui\TableColumn\Status($states));
```

Current list of states supported:

- positive (icon checkmark)
- negative (icon close)
- and the default/unspecified state (icon question)

(list of states may be expanded further)

Template

class atk4\ui\TableColumnTemplate

This column is suitable if you wish to have custom cell formatting but do not wish to go through the trouble of setting up your own class.

If you wish to display movie rating “4 out of 10” based around the column “rating”, you can use:

```
$table->addColumn('rating', new \atk4\ui\TableColumn\Template('{ $rating } out of 10'));
```

Template may incorporate values from multiple fields in a data row, but current implementation will only work if you assign it to a primary column (by passing 1st argument to addColumn).

(In the future it may be optional with the ability to specify caption).

Checkbox

class atk4\ui\TableColumnCheckbox

```
atk4\ui\TableColumnCheckbox::jsChecked()
```

Adding this column will render checkbox for each row. This column must not be used on a field. Checkbox column provides you with a handy jsChecked() method, which you can use to reference current item selection. The next code will allow you to select the checkboxes, and when you click on the button, it will reload \$segment component while passing all the id's:

```
$box = $table->addColumn(new \atk4\ui\TableColumn\Checkbox());
$button->on('click', new jsReload($segment, ['ids'=>$box->jsChecked()]));
```

jsChecked expression represents a JavaScript string which you can place inside a form field, use as argument etc.

Actions

class atk4\ui\TableColumnActions

This column can have number of buttons (or similar views) inside a column. This would allow you to interact with each row directly.

The basic usage format is:

```
$act = $table->addColumn(new \atk4\ui\TableColumn\Actions());
```

Forms

class atk4\ui\Form

One of the most important components of Agile UI is the “Form” - a View:

Example fields added one-by-one

Name

Email

Example of field grouping

Address with label

City

Country

Name

Features of a Form include:

- **Rendering a beautiful and valid form:**
 - wide range of supported field types
 - field grouping (more than one field per line)
 - define field width, positioning and size
 - labels, placeholders and hints
 - supports automated layouts or you can define a custom one
- **Integration with Model objects:**
 - automatically populate all or specific fields
 - handle multi-field validation
 - use of semi-automated layouting (you can arrange group of fields)
 - respect caption and other ui-related settings defined in a model
 - lookup referenced models for data
 - data types are converted automatically (e.g. date, time, boolean)
- **JavaScript integration**
 - form is submitted using JavaScript
 - during submit, the loading indicator is shown
 - javascript sends data through POST
 - POST data is automatically parsed and imported into Model
- **You may define onSubmit PHP handler that:**
 - can handle more validation

- make advanced decisions before saving data
- perform a different Actions, such as reload parts of page or close dialog.
- save data into multiple models
- indicate successful completion of a form through a nicely formatted message
- anything else really!

Creating Basic Forms

To create a form you need the following code:

```
$form = new \atk4\ui\Form();
$form->addField('email');

$app->layout->add($form);
```

The first line creates a “Form” object that is assigned to variable *\$f*. Next line defines a new field that is placed inside a form. Once form is defined, it needs to be placed somewhere in a Render Tree, so that the users can see it.

If *\$form* is not yet associated with a model (like above) then an empty model will be created. If you are not using the model, you will need to define *onSubmit()* handler. (Your other option is to use *setModel()*).

Adding Fields to a form

```
atk4\ui\Form::addField(data_field, form_field = null)
```

Create a new field on a form. The first argument is a data field definition. This can simply be a string “email”. Additionally you can specify an array or even a instance of *atk4dataField*

If form is associated with a model, and the specified field exists, then Data Field object will be looked up. Data Field defines type, caption, possible values and other information about the data itself.

Second argument can be used to describe area around the field, which is a visual object derived from *Form::Field*. The class usually is guessed from the data field type, but you can specify your own object here. Alternatively you can pass array which will be used as defaults when creating appropriate Form Field.

Here are some of the examples:

```
// Data field type decides form field class
$form->addField(['is_accept_terms', 'type'=>'boolean']);

// Specifying enum makes form use drop-down
$form->addField(['agree', 'enum'=>['Yes', 'No']]);

// We can switch to use Radio selection
$form->addField(['agree', 'enum'=>['Yes', 'No']], new \atk4\ui\FormField\Radio());
```

Important: Always use *'type'=>* because this also takes care of *type-casting* e.g. converting data formats.

Integrating Form with a Model

As you work on your application, in most cases you will be linking Form with *Model* <<http://agile-data.readthedocs.io/en/develop/model.html>>. This is much more convenient and takes care of handling data flow all the way from the user input to storing them in the database.

`atk4\ui\Form::setModel($model[, $fields])`

Associate field with existing model object and import all editable fields in the order in which they were defined inside model's `init()` method.

You can specify which fields to import and their order by simply listing field names through second argument.

Specifying “false” or empty array as a second argument will import no fields.

property `atk4\ui\Form::$model`

Model that is currently associated with a Form.

For the next demo, lets actually define a model *Person*:

```
class Person extends \atk4\data\Model
{
    public $table = 'person';

    public function init()
    {
        parent::init();
        $this->addField('name', ['required'=>true]);
        $this->addField('surname');
        $this->addField('gender', ['enum' => ['M', 'F']]);
    }

    public function validate()
    {
        $errors = parent::validate();

        if ($this['name'] == $this['surname']) {
            $errors['surname'] = 'Your surname cannot be same as the name';
        }

        return $errors;
    }
}
```

We can now populate form fields based around the data fields defined in the model:

```
$app->layout->add('Form')
->setModel(new Person($db));
```

This should display a following form:

```
$form->addField('terms', ['type'=>'boolean', 'ui'=>['caption'=>'Accept Terms and Conditions']]
);
```

Form Submit Handling

`atk4\ui\Form::onSubmit($callback)`

Specify a PHP call-back that will be executed on successful form submission.

```
atk4\ui\Form::error($field, $message)
```

Create and return *jsChain* action that will indicate error on a field.

```
atk4\ui\Form::success($title[, $sub_title])
```

Create and return *jsChain* action, that will replace form with a success message.

property atk4\ui\Form::\$successTemplate

Name of the template which will be used to render success message.

To continue with my example, I'd like to add new Person record into the database but only if they have also accepted terms and conditions. I can define onSubmit handler that would perform the check, display error or success message:

```
$form->onSubmit(function($form) {
    if (!$form->model['terms']) {
        return $form->error('terms', 'You must accept terms and conditions');
    }

    $form->model->save();

    return $form->success('Registration Successful', 'We will call you soon.');
```

Callback function can return one or multiple JavaScript actions. Methods such as *error()* or *success()* will help initialize those actions for your form. Here is a code that can be used to output multiple errors at once. I intentionally didn't want to group errors with a message about terms and conditions:

```
$form->onSubmit(function($form) {
    $errors = [];

    if (!$form->model['name']) {
        $errors[] = $form->error('name', 'Name must be specified');
    }

    if (!$form->model['surname']) {
        $errors[] = $form->error('surname', 'Surname must be specified');
    }

    if ($errors) {
        return $errors;
    }

    if (!$form->model['terms']) {
        return $form->error('terms', 'You must accept terms and conditions');
    }

    $form->model->save();

    return $form->success('Registration Successful', 'We will call you soon.');
```

At the time of writing, Agile UI / Agile Data does not come with a validation library, but you can use any 3rd party validation code.

Callback function may raise exception. If Exception is based on `\atk4\core\Exception`, then the parameter "field" can be used to associate error with specific field:

```
throw new \atk4\core\Exception(['Sample Exception', 'field'=>'surname']);
```

If 'field' parameter is not set or any other exception is generated, then error will not be associated with a field. Only

the main Exception message will be delivered to the user. Core Exceptions may contain some sensitive information in parameters or back-trace, but those will not be included in response for security reasons.

Form Layout

When you create a Form object and start adding fields through either `addField()` or `setModel()`, they will appear one under each-other. This arrangement of fields as well as display of labels and structure around the fields themselves is not done by a form, but another object - "Form Layout". This object is responsible for the field flow, presence of labels etc.

`atk4\ui\Form::setLayout (FormLayoutGeneric $layout)`

Sets a custom FormLayout object for a form. If not specified then form will automatically use FormLayoutGeneric.

property `atk4\ui\Form::$layout`

Current form layout object.

`atk4\ui\Form::addHeader ($header)`

Adds a form header with a text label. Returns View.

`atk4\ui\Form::addGroup ($header)`

Creates a sub-layout, returning new instance of a `FormLayoutGeneric` object. You can also specify a header.

class `atk4\ui\FormLayoutGeneric`

Renders HTML outline encasing form fields.

property `atk4\ui\FormLayoutGeneric::$form`

Form layout objects are always associated with a Form object.

`atk4\ui\FormLayoutGeneric::addField()`

Same as `Form::addField()` but will place a field inside this specific layout or sub-layout.

My next example will add multiple fields on the same line:

```
$form->setModel(new User($db), false); // will not populate any fields automatically

$form->addFields(['name', 'surname']);

$gr = $form->addGroup('Address');
$gr->addFields(['address', 'city', 'country']); // grouped fields, will appear on the_
↳same line
```

By default grouped fields will appear with fixed width. To distribute space you can either specify proportions manually:

```
$gr = $f->addGroup('Address');
$gr->addField('address', ['width'=>'twelve']);
$gr->addField('code', ['Post Code', 'width'=>'four']);
```

or you can divide space equally between fields. I am also omitting header for this group:

```
$gr = $f->addGroup(['n'=>'two']);
$gr->addFields(['city', 'country']);
```

You can also use in-line form groups. Fields in such a group will display header on the left and the error messages appearing on the right from the field:

```
$gr = $f->addGroup(['Name', 'inline'=>true]);
$gr->addField('first_name', ['width'=>'eight']);
$gr->addField('middle_name', ['width'=>'three', 'disabled'=>true]);
$gr->addField('last_name', ['width'=>'five']);
```

Semantic UI modifiers

There are many other classes Semantic UI allow you to use on a form. The next code will produce form inside a segment (outline) and will make fields appear smaller:

```
$f = new \atk4\ui\Form(['small segment']);
```

For further styling see documentation on *View*.

Button

class `atk4\ui\Button`

Implements a clickable button:

```
$button = $view->add(new \atk4\ui\Button('Click me'));
```

Button will typically inherit all same properties of a *View*. Functionality of *View* alone yields in many various usage patterns such as:

```
$b1 = new Button(['Load', 'primary']);

$button = new Button('Hello there');
$button->addClass('size big');
```

Icons

property `atk4\ui\Button::$icon`

Includes icon on the button:

```
$bar = new Buttons('vertical'); // NOTE: class called Buttons, not Button
$bar->add(new Button(['Play', 'icon'=>'play']));
$bar->add(new Button(['Pause', 'icon'=>'pause']));
$bar->add(new Button(['Shuffle', 'icon'=>'shuffle']));
```

Icon can also be specified as object:

```
$b1 = new Button(['Forks', 'blue', 'icon'=>new Icon('fork')]);
```

property `atk4\ui\Button::$iconRight`

Setting this will display icon on the right of the button:

```
$b1 = new Button(['Next', 'iconRight'=>'right arrow']);
```

Apart from being on the right, same rules apply as `Button::$icon`. Both icons cannot be specified simultaneously.

Linking

```
atk4\ui\Button::link()
```

Will link button to a destination URL or page:

```
$button->link('http://google.com/');  
// or  
$button->link(['details', 'id'=>123]);
```

If array is used, it's routed to `App::url`

For other JavaScript actions you can use *JavaScript Mapping*:

```
$button->js('click', new jsExpression('document.location.reload()'));
```

Complex Buttons

Knowledge of Semantic UI button (<http://semantic-ui.com/elements/button.html>) can help you in creating more complex buttons:

```
$forks = new Button(['labeled'=> true]); // Button, not Buttons!  
$forks->add(new Button(['Forks', 'blue']))->add(new Icon('fork'));  
$forks->add(new Label(['1,048', 'basic blue left pointing']));  
$layout->add($forks);
```

Paginator

```
class atk4\ui\Paginator
```

Paginator displays a horizontal UI menu providing links to pages when all of the content does not fit on a page. Paginator is a stand-alone component but you can use it in conjunction with other components.

Adding and Using

Place paginator in a designated spot on your page. You also should specify what's the total number of pages paginator should have:

```
$paginator = $layout->add('Paginator');  
$paginator->total = 20;
```

Paginator will not display links to all the 20 pages, instead it will show first, last, current page and few pages around the current page. Paginator will automatically place links back to your current page through `App::url()`.

After initializing paginator you can use it's properties to determine current page. Quite often you'll need to display current page BEFORE the paginator on your page:

```
$h = $page->add('Header');  
$page->add('LoremIpsum'); // some content here  
  
$p = $page->add('Paginator');  
$h->set('Page '.$p->page.' from '.$p->total);
```


Remember that values of ‘page’ and ‘total’ are integers, so you may need to do type-casting:

```
$label->set($p->page); // will not work
$label->set((string)$p->page); // works fine
```

Range and Logic

You can configure Paginator through properties.

Reasonable values for \$range would be 2 to 5, depending on how big you want your paginator to appear. Provided that you have enough pages, user should see $\$range * 2 + 1$ bars.

You can override this method to implement a different logic for calculating which page links to display given the current and total pages.

Returns number of current page.

Template

Paginator uses Semantic UI *ui pagination menu* so if you are unhappy with the styling (e.g: active element is not sufficiently highlighted), you should refer to Semantic UI or use alternative theme.

The template for Paginator uses custom logic:

- *rows* region will be populated with list of page items
- *Item* region will be cloned and used to represent a regular page
- *Spacer* region will be used to represent ‘...’
- *FirstItem* if present, will be used for link to page “1”. Otherwise *Item* is used.
- *LastItem* if present, shows the link to last page. Otherwise *Item* is used.

Each of the above (except *Spacer*) may have *active*, *link* and *page* tags.

Dynamic Reloading

Specifying a view here will cause paginator to only reload this particular component and not all the page entirely. Usually the View you specify here should also contain the paginator as well as possibly other components that may be related to it. This technique is used by *Grid* and some other components.

Miscellaneous Views

There are a lot of Views in Agile Toolkit that all behave similar and can be summarized together. Those views share the following properties:

- Extend directly from View
- Use single HTML element or a very simple element structure
- Do not benefit from setModel other than setting the title
- May contain other misc elements

class Icon

Icon

Probably the simplest element, Icon is often a single element as depicted on <http://semantic-ui.com/elements/icon.html>. To add icon inside your application:

```
$this->add('Icon', 'bomb');
```

The *Seed* is used as an icon class. It may contain spaces to separate multiple classes. You can add additional classes or remove them through `View::addClass` `View::removeClass`:

```
$this->add('Icon', 'flag')->addClass('outline');
```

If you are not ready to add 'Icon' into a render-tree, you can use *new*:

```
$icon = new Icon('battery low');
```

Special Icons

You may add additional classes to your icon for size, positioning, disabling, rotation, decorating. Consult documentation on Semantic UI, since Agile UI will not perform any special treatment on those classes:

```
$icon = new Icon('circular inverted teal users');

$no_users = new View([null, 'huge icons']);
$no_users->add(new Icon('big red dont'));
$no_users->add(new Icon('black user icon'));

$chainsaw = new View([null, 'huge icons']);
$chainsaw->add(new Icon('big loading sun'));
$chainsaw->add(new Icon('user'));
```

Agile UI does not implement a special class for 'Icons' because it has not much custom properties and can easily used through a generic View class. The next example demonstrates how to integrate 'corner icon' into a header:

```
$tw = new View([null, 'large icons']);
$tw->add(new Icon('twitter'));
$tw->add(new Icon('inverted corner add'));

$h = new Header(['Add on Twitter', ['icon'=>$tw]]);
```

As a part of

Icon class is often used as a part of another view. For instance a Button may have an icon defined:

```
$b = new Button(['Learn', 'icon'=>'book']);
```

This format is identical but cleaner than passing the object (also saves the work until rendering phase):

```
$b = new Button(['Learn', 'icon'=>new Icon('book')]);

// or

$button = new Button('Learn');
$button -> icon = 'book';
```

Most other use the following pattern to decode 'icon' property. If you are designing a component that may contain one or several icons, use the following rules:

- define each icon positioning separately. (icon, leftIcon, etc)
- the main icon properly should be caled 'icon'
- icon may be string, array or an object

The code placed in `renderView` will look like this:

```
if ($this->icon) {
    if (!is_object($this->icon)) {
        $this->icon = new Icon($this->icon);
    }
    $this->add($this->icon, 'Icon');
}
```

Following this pattern will make sure that developer who uses your component is able to inject an alternative object for an icon yet has the ability to use short format. Additionally between the `init` and `View::renderView` other logic may intervene and perform actions with the icon.

Some elements will also add an extra class when icon is used, for example when adding 'icon' property with the button, it will add 'labeled' class.

class Label

Label

Implementing <http://semantic-ui.com/elements/label.html> for Agile UI, Label is a very basic view, that can be used on it's own or as part of another UI view (such as menu item).

Basic Usage

First argument of constructor or first element in array passed to constructor is considered:

```
$layout->add(['Label', 'hello world']);
```

Label has the following propeties:

property Label::\$icon

property Label::\$iconRight

property Label::\$image

property Label::\$imageRight

property Label::\$detail

All the above can be string, array (passed to Icon, Image or View class) or an object.

class HelloWorld

HelloWorld

A very basic class that says hello world. This is a manefistation of our component concept. If using other PHP frameworks may require you to create multiple files and spend considerable time creating even a "Hello, World!" app, then we do it in a single line:

```
$app->layout->add('HelloWorld');
```

The component will output “Hello, World!”.

class LoremIpsum

LoremIpsum

This is another component that is included for learning purposes, but is also quite useful in actual development. It saves you a trip to google and some copy-pasting action for a filler text. Simply add ‘LoremIpsum’ component:

```
$app->layout->add('LoremIpsum');
```

You may specify amount of text to be generated with lorem:

```
$app->layout->add(['LoremIpsum', 1]); // just add a little one  
  
// or  
  
$app->layout->add(new LoremIpsum(5)); // adds a lot of text
```

class Columns

Columns

This class implements CSS Grid or ability to divide your elements into columns. If you are an expert designer with knowledge of HTML/CSS we recommend you to create your own layouts and templates, but if you are not sure how to do that, then using “Columns” class might be a good alternative for some basic content arrangements.

Columns::addColumn()

When you add new component to the page it will typically consume 100% width of its container. Columns will break down width into chunks that can be used by other elements:

```
$c = $page->add(new \atk4\ui\Columns());  
$c->addColumn()->add(['LoremIpsum', 1]);  
$c->addColumn()->add(['LoremIpsum', 1]);
```

By default width is equally divided by columns. You may specify a custom width expressed as fraction of 16:

```
$c = $page->add(new \atk4\ui\Columns());  
$c->addColumn(6)->add(['LoremIpsum', 1]);  
$c->addColumn(10)->add(['LoremIpsum', 2]); // wider column, more filler
```

You can specify how many columns are expected in a grid, but if you do you can’t specify widths of individual columns. This seem like a limitation of Semantic UI:

```
$c = $page->add(new \atk4\ui\Columns(['width'=>4]));  
$c->addColumn()->add(new Box(['red']));  
$c->addColumn([null, 'right floated'])->add(new Box(['blue']));
```

Rows

When you add columns for a total width which is more than permitted, columns will stack below and form a second row. To improve and control the flow of rows better, you can specify addRow():

```
$c = $page->add(new \atk4\ui\Columns(['internally celled']));

$r = $c->addRow();
$r->addColumn([2, 'right aligned'])->add(['Icon', 'huge home']);
$r->addColumn(12)->add(['LoremIpsum', 1]);
$r->addColumn(2)->add(['Icon', 'huge trash']);

$r = $c->addRow();
$r->addColumn([2, 'right aligned'])->add(['Icon', 'huge home']);
$r->addColumn(12)->add(['LoremIpsum', 1]);
$r->addColumn(2)->add(['Icon', 'huge trash']);
```

This example also uses custom class for Columns (‘internally celled’) that adds dividers between columns and rows. For more information on available classes, see <http://semantic-ui.com/collections/grid.html>.

Responsiveness and Performance

Although you can use responsiveness with the Column class to some degree, we recommend that you create your own component template where you can have greater control over all classes.

Similarly if you intend to output a lot of data, we recommend you to use `Lister` instead with a custom template.

JavaScript Mapping

A modern user interface cannot exist without JavaScript. Agile UI provides you assistance with generating and executing events directly from PHP and the context of your Views. The most basic example of such integration would be a button, that hides itself when clicked:

```
$b = new Button();  
$b->js('click')->hide();
```

Introduction

Agile UI does not replace JavaScript. It encourages you to keep JavaScript routines as generic as possible, then associate them with your UI through actions and events.

A great example would be *jQuery* library. It is designed to be usable with any HTML mark-up and by specifying selector, you can perform certain actions:

```
$('#my-long-id').hide();
```

Agile UI provides a built-in integration for jQuery. To use jQuery and any other JavaScript library in Agile UI you need to understand how Action sand Events work.

Actions

Action is represented through a PHP object that can map itself into a JavaScript code. For instance the code for hiding a view can be generated by calling:

```
$action = $view->js()->hide();
```

We used this *hide* method to previously hide the button. There are other ways to generate action, such as using `jsExpression()`:

```
$action = new jsExpression('alert([])', ['Hello world']);
```

Finally, actions can be used inside other actions:

```
$action = new jsExpression('alert([])', [  
    $view->js()->text()  
]);  
  
// will produce alert($('#button-id').text());
```

or:

```
$action = $view->js()->text(new jsExpression('[] + []', [  
    5,  
    10  
]));
```

All of the mentioned 4 examples will produce a valid “action” object that can be used further.

Important: We never encourage writing JavaScript logic in PHP. The purpose of JS layer is for binding events and actions with your generic JavaScript routines.

Events

Agile UI also offers a great way to associate your actions with certain client-side events. Those events can be triggered by the user or by other JavaScript code. There are several ways to bind *\$action*.

To execute actions instantly on page load, use *true* as first argument to *js()*:

```
$view->js(  
    true,  
    new jsExpression('alert([])', ['Hello world'])  
);
```

You can also combine both forms together:

```
$view->js(true)->hide();
```

Finally, you can specify name of JavaScript event:

```
$view->js('click')->hide();
```

Agile UI also provides support for an *on* event binding. This allows you to apply events on multiple elements:

```
$buttons = new Buttons();  
  
$buttons->add(new Button('One'));  
$buttons->add(new Button('Two'));  
$buttons->add(new Button('Three'));  
  
$buttons->on('click', '.button')->hide();
```

All the above examples will map themselves into a simple and readable JavaScript code. If you wish to see what JavaScript code is produced by certain view or it’s sub-elements, see debugging

Extending

Agile UI builds upon the concepts of actions and events in the following ways:

- Action can be any arbitrary JavaScript with parameters: - parameters are always escaped with `json_encode`, - action can have another nested actions, - you can build your own integration patterns.
- `jsChain` provide Action extension for JavaScript frameworks: - jQuery is implementation of jQuery binding through `jsChain`, - various 3rd party extensions can integrate other frameworks, - any jQuery plugin will work out-of-the-box.
- PHP closure can be used to wrap action-generation code: - Agile UI event will map AJAX call to the event, - closure can respond with additional actions, - various UI elements (such as Form) extend this concept further.

Including JS/CSS

Sometimes you need to include an additional `.js` or `.css` file for your code to work. See `App::includeJS()` and `App::includeCSS()` for details.

Building actions with jsExpressionable

interface jsExpressionable

Allow objects of the class implementing this interface to participate in building JavaScript expressions.

`jsExpressionable::jsRender()`

Express object as a string containing valid JavaScript statement or expression.

`View` class implements `jsExpressionable` and will present itself as a valid selector. Example:

```
$frame = new View();

$button->js(true)->appendTo($frame);

// Resulting code:
// $('#button-id').appendTo('#frame-id');
// which will be executed on page load
```

JavaScript Chain Building

class jsChain

Base class `jsChain` can be extended by other classes such as jQuery to provide transparent mappers for any JavaScript framework.

Chain is a PHP object that represents one or several actions that are to be executed on the client side. The `jsChain` objects themselves are generic, so in my examples I'll be using jQuery which is a descendant of `jsChain`:

```
$chain = new jQuery('#the-box-id');

$chain->dropdown();
```

The calls to the chain are stored in the object and can be converted into JavaScript by calling `jsChain::jsRender()`

`jsChain::jsRender()`

Converts actions recorded in `jsChain` into string of JavaScript code.

Executing:

```
echo $chain->jsRender();
```

will output:

```
$('#the-box-id').dropdown();
```

Important: It's considered a vary bad practice if you perform `jsRender` and output the JavaScript code manually. Agile UI takes care of JavaScript binding and also decides which actions should be appearing for you as long as you create actions for your chain.

`jsChain::_json_encode()`

`jsChain` will map all the other methods into JS counterparts while encoding all the arguments through `_json_encode()`. Although similar to a standard `json_encode` function, this method recognizes `jsExpressionable` objects and will substitute them with the result of `jsExpressionable::jsRender`. The result will not be escaped and any object implementing `jsExpressionable` interface is responsible for safe JavaScript generation.

The following code is safe:

```
$b = new Button();  
$b->js(true)->text($_GET['button_text']);
```

Any malicious input through the GET arguments will be wrapped through `json_encode` before being included as an argument to `text()`.

View to JS integration

We are not building JavaScript code just for the exercise. Our whole point is ability to link that code between actual views. All views support JavaScript binding through two methods: `View::js()` and `View::on()`.

class View

`View::js([$event[, $other_action]])`

Return action chain that targets this view. As event you can specify `true` which will make chain automatically execute on document ready event. You can specify a specific JavaScript event such as “*click*” or “*mousein*”. You can also use your custom event that you would trigger manually. If `$event` is false or null, no event binding will be performed.

If `$other_chain` is specified together with event, it will also be bound to said event. `$other_chain` can also be a PHP closure.

Several usage cases for plain `js()` method. The most basic scenario is to perform action on the view when event happens:

```
$b1 = new Button('One');  
$b1->js('click')->hide();  
  
$b2 = new Button('Two');  
$b2->js('click', $b1->js()->hide());
```

`View::on(String $event[, String selector], $callback = null)`

Returns chain that will be automatically executed if `$event` occurs. If `$callback` is specified, it will also be executed on event.

The following code will show 3 buttons and clicking any button will hide itself. Only a single action is created:

```
$buttons = Buttons();

$buttons->add(new Button('One'));
$buttons->add(new Button('Two'));
$buttons->add(new Button('Three'));

$buttons->on('click', '.button')->hide();

// Generates:
// $('#top-element-id').on('click', '.button', function($event){
//     event.stopPropagation();
//     event.preventDefault();
//     $(this).hide();
// });
```

Method `on()` is handy when you have multiple elements inside your view that you want to trigger action individually. The best example would be a `Listner` with interactive elements:

```
$buttons = Buttons();

$b1 = $buttons->add(new Button('One'));
$b2 = $buttons->add(new Button('Two'));
$b3 = $buttons->add(new Button('Three'));

$buttons->on('click', '.button', $b3->js()->hide());

// Generates:
// $('#top-element-id').on('click', '.button', function($event){
//     event.stopPropagation();
//     event.preventDefault();
//     $('#b3-element-id').hide();
// });
```

You can use both actions together. The next example will allow only one button to be active:

```
$buttons = Buttons();

$b1 = $buttons->add(new Button('One'));
$b2 = $buttons->add(new Button('Two'));
$b3 = $buttons->add(new Button('Three'));

$buttons->on('click', '.button', $b3->js()->hide());

// Generates:
// $('#top-element-id').on('click', '.button', function($event){
//     event.stopPropagation();
//     event.preventDefault();
//     $('#b3-element-id').hide();
// });
```

jsExpression

class jsExpression

`jsExpression::__construct (template, args)`

Returns object that renders into template by substituting args into it.

Sometimes you want to execute action by calling a global JavaScript method. For this and other cases you can use `jsExpression`:

```
$action = new jsExpression('alert([])', [
    $view->js()->text()
]);
```

Because `jsChain` will typically wrap all the arguments through `jsChain::_json_encode()`, it prevents you from accidentally injecting a JavaScript code:

```
$b = new Button();
$b->js(true)->text('2+2');
```

This will result in a button having a label `2+2` instead of having a label `4`. To get around this, you can use `jsExpression`:

```
$b = new Button();
$b->js(true)->text(new jsExpression('2+2'));
```

This time `2+2` is no longer escaped and will be used as a plain JS code. Another example shows how you can use global variables:

```
echo (new jQuery('document'))->find('h1')->hide()->jsRender();

// produces $('document').find('h1').hide();
// does not hide anything because document is treated as string selector!

$expr = new jsExpression('document');
echo (new jQuery($expr))->find('h1')->hide()->jsRender();

// produces $(document).find('h1').hide();
// works correctly!!
```

Template of jsExpression

The `jsExpression` class provides the most simple implementation that can be useful for providing any JavaScript expressions. My next example will set height of right container to the sum of 2 boxes on the left:

```
$h1 = $left_box1->js()->height();
$h2 = $left_box2->js()->height();

$sum = new jsExpression('[ ]+[ ]', [$h1, $h2]);

$right_box_container->js(true)->height($sum);
```

It is important that you remember that height of an element is a browser-side property and you must operate with it in your browser by passing expressions into chain.

The template language for `jsExpression` is super-simple:

- `[]` will be mapped to next argument in the argument array
- `[foo]` will be mapped to named argument in argument array

So the following three lines are identical:

```
$sum = new jsExpression('[0]+[1]', [$h1, $h2]);
$sum = new jsExpression('[a]+[b]', ['a'=>$h1, 'b'=>$h2]);
```

Important: We have specifically selected a very simple tag format as a reminder to you not to write any code as part of `jsExpression`. You must not use `jsExpression()` for anything complex.

Writing JavaScript code

If you know JavaScript you are likely to write more extensive methods to provide extended functionality for your user browsers. Agile UI does not attempt to stop you from doing that, but you should follow a proper pattern.

Open a new file `test.js` and type:

```
function mySum(arr) {
    return arr.reduce(function(a, b) {
        return a+b;
    }, 0);
}
```

Then load this JavaScript dependency on your page. Refer to `App::includeJS()` and `App::includeCSS()`. Finally use UI code as a “glue” between your routine and the actual View objects. In my example, I’ll be trying to match the size of `$right_container` with the size of `$left_container`:

```
$heights = [];

foreach ($left_container->elements as $left_box) {
    $heights[] = $left_box->js()->height();
}

$right_container->js(true)->height(new jsExpression('mySum([], [$heights])');
```

This will map into the following JavaScript code:

```
$('#right_container_id').height(mySum([
    $('#left_box1').height(), $('#left_box2').height(), $('#left_box3').height() //
    ↪ etc
]));
```

You can further simplify JavaScript code yourself, but keep the JavaScript logic inside the `.js` files and leave PHP only for binding.

Reloading

class jsReload

`jsReload` is a JavaScript action that performs reload of a certain object:

```
$js_reload_table = new jsReload($table);
```

This action can be used similar to any other `jsExpression`. For instance completing the form can reload some other view:

```
$m_book = new Book($db);

$f = $app->layout->add('Form');
$t = $app->layout->add('Table');

$f->setModel($m_book);

$f->onSubmit(function($f) use($t) {
    $f->model->save();
    return new \atk4\ui\jsReload($t);
});

$t->setModel($m_book);
```

In this example, filling out and submitting the form will result in table contents being refreshed using AJAX.

Agile Data

Agile Data is a business logic and data persistence framework. It's a separate library that has been specifically designed and developed for use in Agile UI.

With Agile Data you can easily connect your UI with your data and make UI components store your data in SQL, NoSQL or RestAPI. On top of the existing persistences, Agile UI introduces a new persistence class: "UI".

This UI persistence will be extensively used when data needs to be displayed to the user through UI elements or when input must be received from the UI layer.

If you do not intend to store data anywhere or are using your own ORM, the Agile Data will still be used to some extent and therefore it appears as requirement.

Most of the ORMs lack several important features that are necessary for UI framework design:

- ability to load/store data safely with conditions.
- built-in support for column meta-information
- field, type and table mapping
- "onlyFields" support for efficient querying
- domain-level model references.

Agile Data is distributed under same open-source license as Agile UI and the rest of this documentation will assume you are using Agile Data for the purpose of overall clarity.

Interface Stability

Agile UI is based on Agile Toolkit 4.3 which has been a maintained UI framework that can trace it's roots back to 2003. As a result, the object interface is highly stable and all of the documented methods, models and properties will not change even in the major releases.

If we do have to change something we will keep things backwards compatible for a period of a few years.

We expect you to extend base classes to build your UI as it is a best practice to use Agile UI.

Testing and Enterprise Use

Agile UI is designed with corporate use in mind. The main aim of the framework is to make your application consistent, modern and fast.

We understand the importance of testing and all of the Agile UI components come fully tested across multiple browsers. In most cases browser compatibility is defined by the underlying CSS framework.

With Agile UI we will provide you with a guide how to test your own components.

Unit Tests

You only need to unit-test your own classes and controllers. For example if your application creates a separate class that deals with APR calculation, you need to include unit-test for that specific class.

Business Logic Unit Tests

Those tests are most suitable for testing your business logic, that is included in Agile Data. Use “array” persistences to pre-set model with the necessary data, execute your business logic with mock objects.

1. set up mock database arrays
2. instantiate model(s)
3. execute business operation
4. assert new content of array.

In most cases the Integration tests are easier to make, and give you equal testability.

Integration Database Tests

This test-suite will operate with SQL database by executing various database operations in Agile Data and then asserting business logic changes.

1. load “safe” database schema
2. each test starts transaction and is finished with a roll-back.
3. perform changes such as adding new invoice
4. assert through other models e.g. by running client report model.

Component Tests

All of the basic components are tested for you using UI tests, but you should test your own components. This test will place your component under various configurations and will make sure that it continues to work.

If your component relies on a model, this can also attempt various model combinations for an extensive test.

User Testing

Once you place your components on your pages and associate them with your actual data you can perform user tests.

base-components core layouts building-components menu lister table paginator grid form crud tree
virtual-page console

CHAPTER 7

Indices and tables

- `genindex`
- `search`

a

`atk4\ui`, 50

Symbols

__clone() (Template method), 31
 __construct() (Template method), 30
 __construct() (View method), 44
 __construct() (jsExpression method), 77
 _json_encode() (jsChain method), 76

A

add() (View method), 42
 addClass() (View method), 44
 addColumn() (atk4\ui\Table method), 52
 addColumn() (Columns method), 70
 addField() (atk4\ui\Form method), 61
 addField() (atk4\ui\FormLayoutGeneric method), 64
 addGroup() (atk4\ui\Form method), 64
 addHeader() (atk4\ui\Form method), 64
 always_run (App property), 20
 App (class), 17
 app (View property), 43
 append() (Template method), 32
 appendHTML() (Template method), 32
 atk4\ui (namespace), 48, 50, 59, 65, 66

B

Button (class in atk4\ui), 65

C

catch_exception (App property), 20
 caughtException() (App method), 20
 class (View property), 44
 columnFactory() (atk4\ui\Table method), 52
 columns (atk4\ui\Table property), 52
 Columns (class), 70
 content (View property), 47

D

default_column (atk4\ui\Table property), 52
 defaultTemplate() (Template method), 35
 del() (Template method), 34

detail (Label property), 69

E

eachTag() (Template method), 34
 encodeAttribute() (App method), 21
 encodeHTML() (App method), 21
 error() (atk4\ui\Form method), 62

F

form (atk4\ui\FormLayoutGeneric property), 64
 Form (class in atk4\ui), 59
 FormLayoutGeneric (class in atk4\ui), 64

G

getCellHTML() (atk4\ui\TableColumnGeneric method), 51
 getHeaderCellHTML() (atk4\ui\TableColumnGeneric method), 51
 getHTML() (View method), 45
 getHTMLTags() (atk4\ui\TableColumnGeneric method), 52
 getJS() (View method), 45
 getJSID() (View method), 47
 getRequestURI() (App method), 20
 getTag() (App method), 21
 getTotalsCellHTML() (atk4\ui\TableColumnGeneric method), 51
 Grid (class in atk4\ui), 48

H

HelloWorld (class), 69

I

icon (atk4\ui\Button property), 65
 Icon (class), 67
 icon (Label property), 69
 iconRight (atk4\ui\Button property), 65
 iconRight (Label property), 69
 image (Label property), 69

imageRight (Label property), **69**
init() (View method), **42**
initDefaultApp() (View method), **48**
initIncludes() (App method), **20**
is_rendering (App property), **20, 21**
isSet() (Template method), **34**

J

js() (View method), **76**
jsChain (class), **75**
jsChecked() (atk4\ui\TableColumnCheckbox method), **59**
jsExpression (class), **77**
jsExpressionable (interface), **75**
jsReload (class), **79**
jsRender() (jsChain method), **75**
jsRender() (jsExpressionable method), **75**

L

Label (class), **69**
layout (atk4\ui\Form property), **64**
link() (atk4\ui\Button method), **66**
load() (Template method), **30**
loadTemplate() (Template method), **31**
loadTemplateFromString() (Template method), **30, 31**
LoremIpsum (class), **70**

M

model (atk4\ui\Form property), **62**
model (View property), **43**

N

name (View property), **47**

O

on() (View method), **76**
onSubmit() (atk4\ui\Form method), **62**
original_filename (Template property), **32**

P

Paginator (class in atk4\ui), **66**

R

recursiveRender() (View method), **48**
region (View property), **46**
reload() (Template method), **31**
removeClass() (View method), **44**
render() (Template method), **33**
render() (View method), **45**
renderView() (View method), **46**
requireCSS() (App method), **20**
requireJS() (App method), **20, 22**
run() (App method), **20**
run_called (App property), **20**

S

set() (Template method), **32**
set() (View method), **48**
setHTML() (Template method), **32**
setLayout() (atk4\ui\Form method), **64**
setModel() (atk4\ui\Form method), **62**
setModel() (View method), **43**
setProperties() (View method), **48**
setProperty() (View method), **48**
skin (View property), **47**
sort_by (atk4\ui\Table property), **53**
sort_order (atk4\ui\Table property), **53**
sortable (atk4\ui\Table property), **53**
stickyForget() (App method), **21**
stickyGet() (App method), **21**
success() (atk4\ui\Form method), **63**
successTemplate (atk4\ui\Form property), **63**

T

Table (class in atk4\ui), **52**
TableColumnActions (class in atk4\ui), **59**
TableColumnCheckbox (class in atk4\ui), **59**
TableColumnGeneric (class in atk4\ui), **51, 57**
TableColumnLink (class in atk4\ui), **58**
TableColumnMoney (class in atk4\ui), **58**
TableColumnStatus (class in atk4\ui), **58**
TableColumnTemplate (class in atk4\ui), **59**
tags (Template property), **32**
Template (class), **29**
template (Template property), **32**
template (View property), **46**
template_source (Template property), **32**
terminate() (App method), **21**
tryDel() (Template method), **34**
trySet() (Template method), **34**

U

ui (View property), **44**
url() (App method), **22**

V

View (class), **41, 76**