Agile UI

Release

Jan 15, 2018

Contents

Contents:

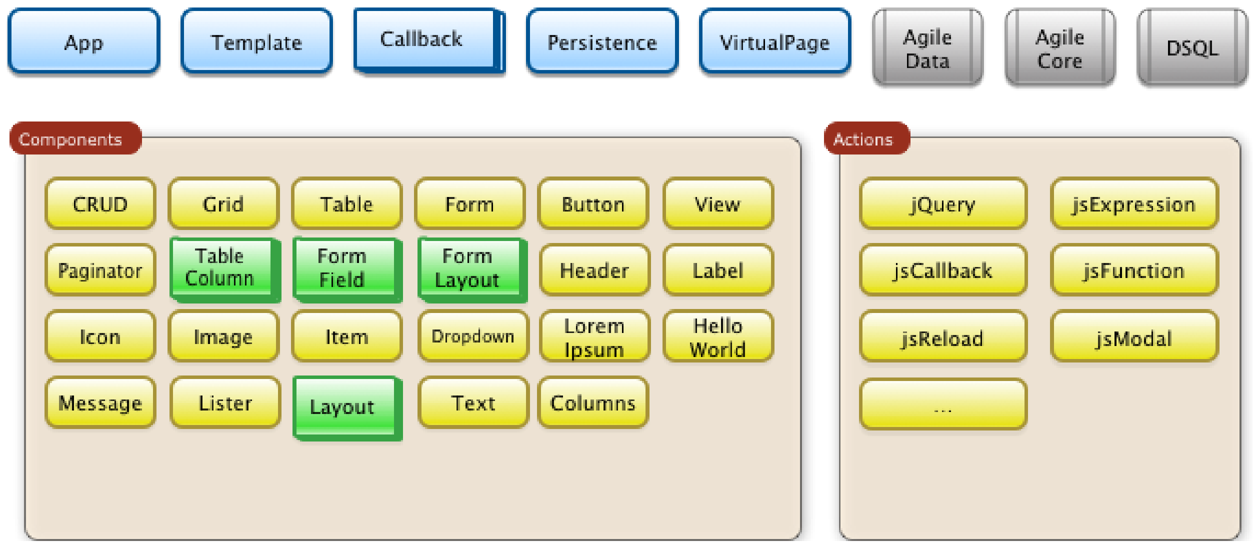
CHAPTER 1

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 use 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 abstricts 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.

1.1 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.

1.1.1 1. Out of the box experience

Sample scenario:

> If during .COM boom you purchased 1000 good-looking .COM domains and are now selling them, you need to track offers from buyers. 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 a scenario. By simply describing your data model, relations and operations, you get a fully working UI and API with minimal setup.

1.1.2 2. Compact and easy to integrate

Simple scenario:

> Your domains such as "happy.com" receive 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 an 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.

1.1.3 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 the same operations.

1.1.4 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.

1.1.5 5. High-level Solution

Simple scenario:

You are a busy person, who needs to get your application ready in 1h and then will 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 will be quick, and, just works.

1.1.5.1 Overview Example

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

```php
<?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-interractive CRUD
$db = new \atk4\data\Persistence_SQL($dsn);
$app->add(new \atk4\ui\CRUD())
    ->setModel(new Offer($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 framework capabilities.

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

  For full documentation see *Purpose of App class*.

- *$db* this is a 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* 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 specific persistence and conditions.

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

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

- *CRUD* is a *Component* 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 the 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 more technical terms, Agile UI:

- Is full of abstraction of Web technologies through components.
- Has concise syntax to define UI layouts in PHP.
- Has built-in security and safety.
- Decouples from data storage/retrievel mechanism.
- And is designed to be integrated into full-stack frameworks.
- Abstains from duplicating field names, types or validation logic outside of Model class.

1.1.6 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.

1.2 Component

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 a component work.

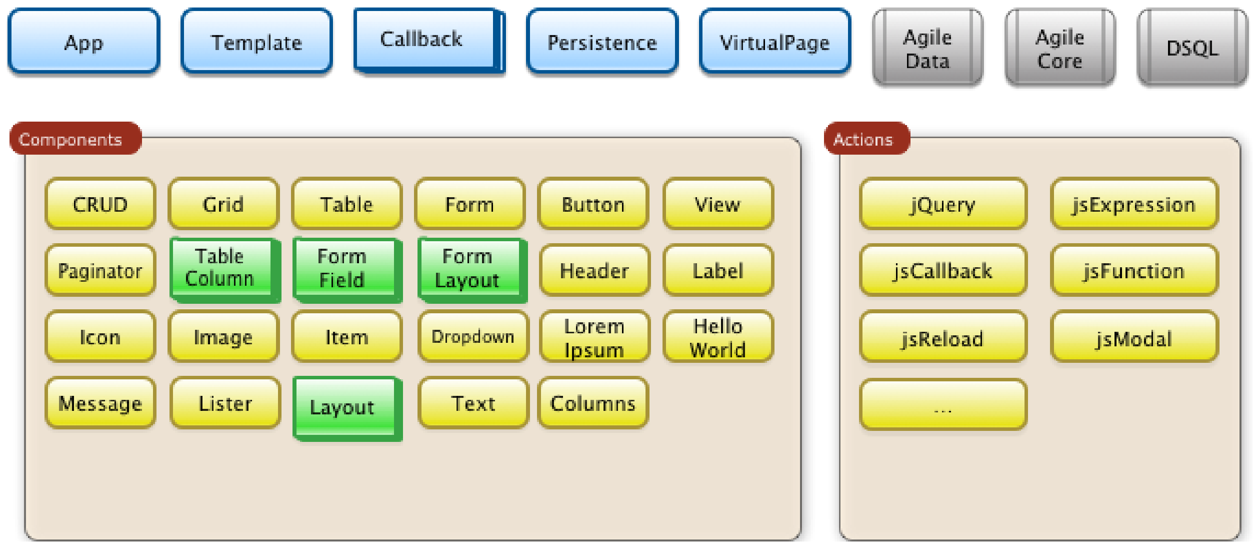
That means that components may rely on each other and even though some may appear very basic to you, they are relied on by some other components for maximum felxibility. The next example adds a "Cancel" button to a form:

```
$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:

1.2.1 Using Components

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



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

To render a 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()`

1.2.2 Factory

Factory is a mechanism which allow you to use shorter syntax for creating objects. The Agile UI goal is to be simple to use and is 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');
```

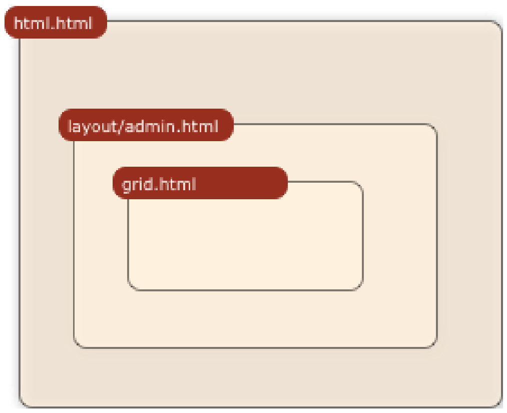
By default, classes specified as 1st element of array passed to the add() method are resolved to namespace *atk4\ui*, however the application class can fine-tune the search.

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

1.2.3 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.

1.2.4 Layouts

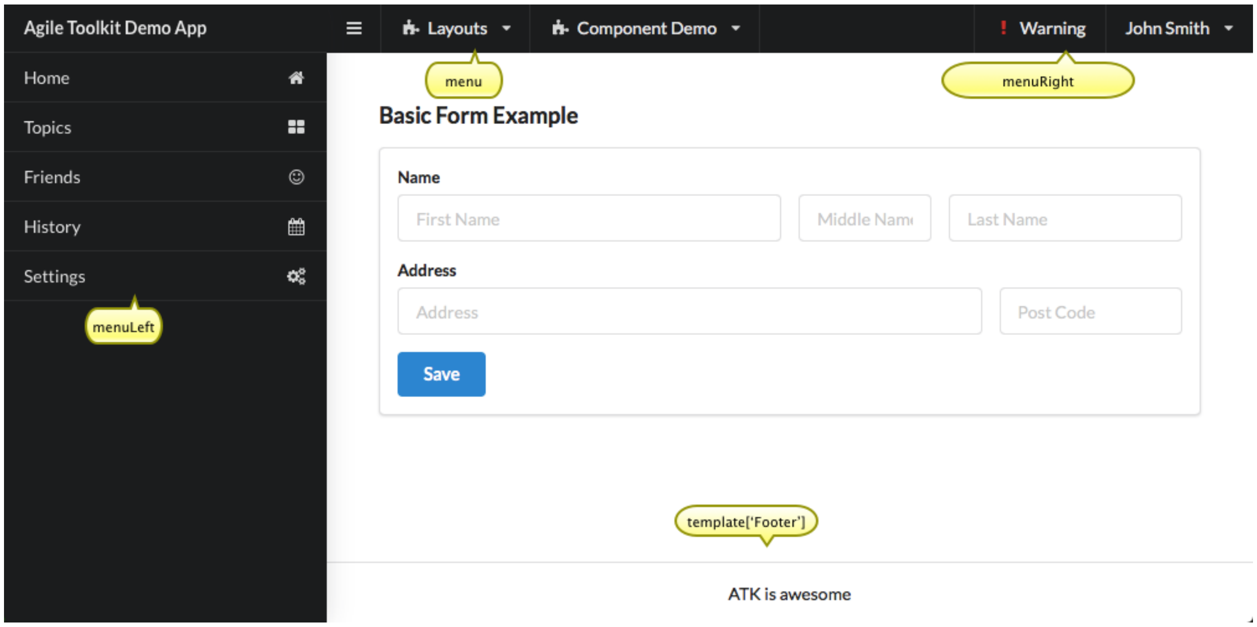


Using App class will utilise a minimum of 2 templates:

- 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 a 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.

1.3 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:

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

1.3.1 Non-PHP dependencies

Your componet 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`

1.3.2 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.

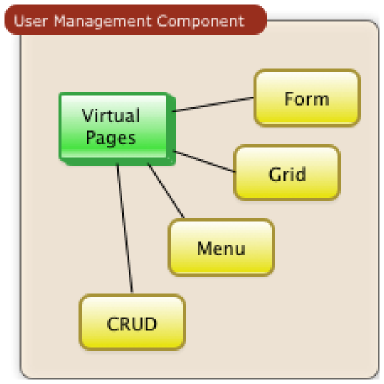
1.3.3 Callbacks

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

Agile UI allows for a 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 *Callbacks*

1.3.4 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.

1.3.5 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 step further by offering you a directory of published add-ons and install them by using a simple wizard.

1.4 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 very 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 stardard UI widgets.

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

1.4.1 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).

1.4.2 Application Tutorials

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

1.4.3 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.

1.4.4 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.

1.5 Things Agile UI simplifies

Some technologies are “prerequirements” 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.

1.5.1 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.

1.5.2 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.

CHAPTER 2

Quickstart

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.

2.1 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.

2.2 Installing

Create a directory wich is accessible by you web server. Start your command-line, enter this directory and execute composer command:

```
composer require atk4/ui
```

2.3 Coding "Hello, World"

Open a new file *index.php* and enter the following code:

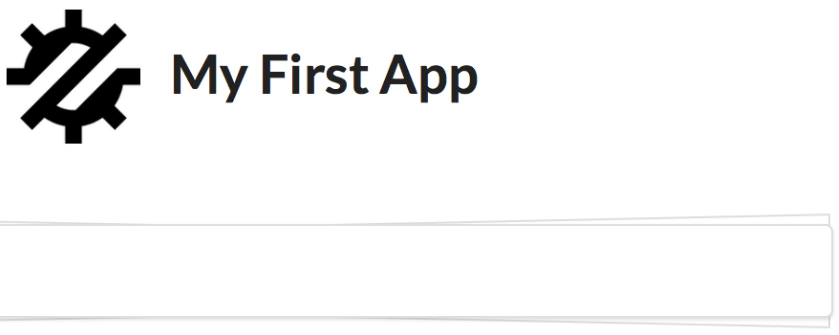
```php
<?php                                   // 1
require 'vendor/autoload.php';          // 2

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

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

Clarifications

You should see the following output:



Instead of manually outputing 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.

2.4 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.

To be able to actually run this example, create a new file todo.php in the same directory as index.php and create the application:

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

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

All components of Agile Data are database-agnostic and will not concern themselve 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);
```

2.5 Data Model

We need a class *Task* which decribes 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.

2.6 Form and CRUD Components

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

```
$col = $app->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.

2.7 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
]]);
$grid->setModel(new ToDoItem($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) {
        $grid->model->delete($id);                       // 25
    }
}
```

Clarifications

2.8 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->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.

2.9 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)

CHAPTER 3

Core Concepts

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.

3.1 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, however the common pattern is:

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

3.1.1 Purpose of App class

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 an App class yourself before other components:

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

As you add one component into another, they will automatically inherit reference to App class. App class is an 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 asumptions about the App class, to keep overal 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 the error is not repeated.

3.1.1.1 Using App for Injecting Depedencies

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.

3.1.1.2 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

3.1.1.3 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 a 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->add(['Message', 'Login Required', 'error']);
            $this->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 a populating menu.
    }
}
```

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

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

3.1.1.4 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->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()*

3.1.1.5 Clean-up and simplification

App::**run**()

property App::$**run_called**

property App::$**is_rendering**

property App::$**always_run**

App 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->add('HelloWorld');

// Next line is optional
$app->run();
```

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

3.1.1.6 Exception handling

App::**caugthException**()

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 a try-catch block.

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

3.1.1.7 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 the current page. See `App::url()`

3.1.2 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.

3.1.2.1 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 interractive, 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 implements will always address same model scope.

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

3.1.2.2 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.
```

3.1.2.3 Execution state

property App::$**is_rendering**

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

3.1.2.4 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*.

3.1.2.5 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.

3.1.2.6 Hooks

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

- beforeRender
- beforeOutput

3.1.3 Application and Layout

When writing an 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 populare paginator
$grid->addButton('Test');       // initialize and populate toolbar

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

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

3.1.3.1 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.

3.1.3.2 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 overal 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');
// or simply which does the same thing
$app->menu->addItem('User Admin', 'admin');
```

3.1.3.3 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.

3.1.3.4 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 of 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.

3.2 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->add(new \atk4\ui\Button('Hello'));

and

$app->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.

3.2.1 Purpose of the 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 differenttly, 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
```

3.2.2 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
```

3.2.3 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-existant property will result in exception:

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

3.2.4 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');
```

3.3 Render Tree

Agile Toolkit is allows you to create components hierarchically. Once complete, the component hierarchy will render itself and will present HTML output that would appear to user.

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'));

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

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

3.3.1 Introduction

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 asigned 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 initialied. 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.

3.3.2 Initialization

Calling init() method of a view is essential before any meaningfull 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 asigned.

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.

3.3.3 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 initalized!
```

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

```
$b = $app->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:

```
$app->add($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.

3.3.4 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 furtile 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.

3.3.5 Unique Name

Through adding objects into render tree (even if those are not Views) objects can assume unique names. When you create your application, then any object you add into your app will have a unique *name* property:

```
$b = $app->add('Button');
echo $b->name;
```

The other property of the name is that it's also "permanent". Refreshing the page guarantees your object to have the same name. Ultimatly, you can create a View that uses it's name to store some information:

```
class MyView extends View {
    function init() {
        parent::init();

        if ($_GET[$this->name]) {
            $this->add(['Label', 'Secret info is', 'big red', 'detail'=>$_GET[$this->
↪name]]);
        }

        $this->add(['Button', 'Send info to ourselves'])
            ->link([$this->name => 'secret_info']);
    }
}
```

This quality of Agile UI objects is further explored thorugh `Callback` and `VirtualPage`

3.4 Type Presentation

Several components are too complex to be implemented in a single class. `Table`, for example, has ability to format columns by utilising type-specific column classes. Another example is `Form` which relies on Field-specific FormField component.

Agile UI uses a specific pattern for those definitions, which makes overal structure more extensible by having an ability to introduce new types with consistent support throughout the UI.

3.4.1 Formatters vs Decorators

This chapter describes a common technique used by various components that wish to preserve extensible nature when dealing with used-defined types. Reading this chapter will also help you understand some of the thinking behind major decisions when designing the type system.

When looking into the default money field in Agile UI, which does carry amount, but not the currency, there are a number of considerations when dealing with the field. The first important concept to understand is distinction between data Presentation and Decoration.

- Data Presentation - displaying value of the data in a different format, e.g. 123,123.00
- Data Decoration - adding currency symbol or calendar icon.

Agile UI believes that Data Presentation must be consistent throughout the system. A monetary field will use same format on the `Form`, `Table` and even inside a custom HTML template specified into generic `View`.

When it comes to decoration, the method is very dependent on the context. A form may present Calendar (DatePicker) or enable field icon to indicate currency.

Presentation in Agile Toolkit is handled by `Persistence\UI` but decoration is done through helper classes, such as `FormField\Calendar` or `TableColumn\Money`.

Decorator is in control of the final output, so it can decide if it should use value from Presentation or do some decoration on its own.

3.4.1.1 Extending Data Types

If you are looking to add a new data type, such as "money+currency" combination, which would allow user to specify both the currency and the monetary value, you should start by adding support for a new type.

In the below steps, the #1 and #2 are a minimum to achieve. #3 and #4 will improve experience of your integration.

1. **Extend UI persistence and use your class prototype in *$app->persistence*. You need to** define how to output your data as well as read it.
2. **Try your new type with a standard Form field. The value you output should read and stored** back correctly. This ensures that standart UI will work with your new data type.
3. **Create your new decorator, such as use drop-down to select currency from a pre-defined list** inside your specific class while extending `FormField\Input` class. Make sure it can interpret input correctly. The process is explained further down in this chapter.
4. **Associate the types with your decorator in `Form::_fieldFactory` and** `Table::_columnFactory`

For the 3rd party add-ons it is only possible to provide decorators. They must rely on one of the standard type, unless they also offer a dedicated model.

3.4.1.2 Manualy Specifying Decorators

When working with components, they allow you to specify decoratorors manually, even if type of your field does not seem compatible:

```
$table->addColumn('field_name', new \atk4\ui\TableColumn\Password());

// or

$form->addField('field_name', new \atk4\ui\FormField\Password());
```

Here is the priority list when selecting decorator:

- specified to addColumn / addField() as second argument
- specified through $field->ui['form'] = new atk4uiFormFieldPassword();
- fallback to Form::_fieldFactory

3.4.2 Use-case scenarios Explained

Here I'm collecting various use-cases and how to properly deal with scenarios

3.4.2.1 Display password in plain-text for Admin

Normally password is presented as asterisks on the Grid and Form. But what if you want to show it without masking just for the admin? Change type in-line for the model field:

```
$m = new User($app->db);
$m->getElement('password')->type = 'string';

$crud->setModel($m);
```

Note: Changing element's type to string will certainly not perform any password encryption.

3.4.2.2 Hide "account_number" on specific Table

This is reverse scenario. Field *account_number* needs to be stored as-is but should be hidden when presented. To hide it from Table:

```
$m = new User($app->db);

$table->setModel($m);
$m->addDecorator('account_number', new \atk4\ui\TableColumn\Password());
```

3.4.2.3 Create a decorator for hiding credit-card number

If you happen to store card numbers and you only want to display last digits when field appears on the tables, yet make it available when editing, you could create your own TableColumn decorator:

```
class Masker extends \atk4\ui\TableColumn\Generic
{
    public function getDataCellTemplate(\atk4\data\Field $f = null)
    {
        return '**** **** **** {$mask}';
    }

    public function getHTMLTags($row, $field)
    {
        return [
            'mask' => substr($field->get(), -4)
        ];
    }
}
```

If you are wondering, why I'm not overriding by providing HTML tag equal to the field name, it's because this technique is unreliable due to ability to exclude HTML tags with `Table::$use_html_tags`.

3.4.2.4 Display card number with spaces

If we always have to display card numbers with spaces, e.g. "1234 1234 1234 1234" but have database store them without spaces, then this is data formatting task. Best done if we extend `Persistence\UI`:

```
class MyPersistence extends \atk4\ui\Persistence\UI
{

    public function _typecastSaveField(\atk4\data\Field $f, $value)
    {
        switch ($f->type) {
        case 'card':
            $parts = str_split($value, 4);
            return join(' ', $parts);
        }
        return parent::_typecastSaveField($f, $value);
    }

    public function _typecastLoadField(\atk4\data\Field $f, $value)
    {
        switch ($f->type) {
        case 'card':
            return str_replace(' ', '', $value);
        }
        return parent::_typecastLoadField($f, $value);
    }
}


class MyApp extends App
{
    public function __construct($defaults = [])
    {
        $this->ui_persistence = new MyPersistence()

        parent::__construct($defaults);
    }

}
```

Now your ‘card’ type will work system-wide.

3.5 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:

3.5.1 Introduction

Agile UI relies on a lightweigt 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

3.5.2 Example Template

Assuming that you have the following template:

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

3.5.2.1 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 idetnical to:

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

3.5.2.2 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 deligate 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());
```

3.5.2.3 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';

$lister = new Lister($v, 'Content');
$lister->setModel($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. Lister will clone region 'Content' from my_template.
4. Lister will associate itself with provided model.
5. When rendering is executed, lister will iterate through the data, appending value of the rendered region back to $v. Finally the $v will render itself and echo result.

3.5.3 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

3.5.3.1 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.

3.5.3.2 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.

3.5.3.3 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`.

3.5.3.4 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	`Lister` will duplicate Region
Associate nested tags with models record	*`View`* with setModel() can do that
Various cases within templates based on condition	cloneRegion or get, then use set()
Custom handling certain tags or regions	`GiTemplate::eachTag` 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 eachTag()

3.5.4 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.

3.5.4.1 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`.

3.5.4.2 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']);
}
```

3.5.4.3 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"
```

3.5.4.4 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 appropriatelly cloned from regions of $envelope and then substituted back after render.

In this example I've usd 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.

3.5.4.5 Other opreations 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.

3.5.4.6 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

3.5.5 Views and Templates

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

3.5.5.1 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
}
```

3.5.5.2 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'));
```

3.5.5.3 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.

3.5.5.4 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);
```

3.5.5.5 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]);
});
```

3.5.6 Best Practices

3.5.6.1 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)

3.5.6.2 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.

3.5.6.3 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".

3.5.7 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.

3.5.8 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 it’s actual name and the name with unique “#1” appended (in case there are multiple instances of same tag). This allow `$smlite->get()` to quickly retrieve contents of appropriate tag and it will also allow `render()` to reconstruct the output efficiently.

3.6 Agile Data

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 worry over the details:

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

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

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

3.6.1 Integration

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

Other benefits of relying on Agile Data models is the 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 it's own resulting in much bigger code footprint.

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

3.6.2 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 its performance will generally be slower too.

3.6.3 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.

3.7 Callbacks

By relying on the ability of generating *Unique Name*, it's possible to create several classes for implementing PHP call-backs. They follow the pattern:

- present something on the page (maybe)
- generate URL with unique parameter
- if unique parameter is passed back, behave differently

Once the concept is established, it can even be used on a higher level, for example:

```
$button->on('click', function() { return 'clicked button'; });
```

3.7.1 Callback Introduction

Agile UI pursues a goal of creating a full-featured, interractive, user interface. Part of that relies on abstraction of Browser/Server communication.

Callback mechanism allow any *Component* of Agile Toolkit to send HTTP requests back to itself through a unique route and not worry about accidentally affecting or triggering action of any other component.

One example of this behaviour is the format of `View::on` where you pass 2nd argument as a PHP callback:

```
$button = new Button();

// clicking button generates random number every time
$button->on('click', function($action){
    return $action->text(rand(1,100));
});
```

This creates call-back route transparently which is triggered automatically during the 'click' event. To make this work seamlessly there are several classes at play. This documentation chapter will walk you through the callback mechanisms of Agile UI.

3.7.2 The Callback class

class Callback

Callback is not a View. This class does not extend any other class but it does implement several important traits:

- TrackableTrait
- AppScopeTrait
- DIContainerTrait

To create a new callback, do this:

```
$c = new \atk4\ui\Callback();
$app->add($c);
```

Because 'Callback' is not a View, it won't be rendered. The reason we are adding into *Render Tree* is for it to establish a unique name which will be used to generate callback URL:

`Callback::`**`getURL`**`(`*$val*`)`

`Callback::`**`set`**`()`

The following example code generates unique URL:

```
$label = $app->add(['Label','Callback URL:']);
$cb = $label->add('Callback');
$label->detail = $cb->getURL();
$label->link($cb->getURL());
```

I have assigned generated URL to the label, so that if you click it, your browser will visit callback URL triggering a special action. We haven't set that action yet, so I'll do it next with :php:meth::*Callback::set()*:

```
$cb->set(function() use($app) {
    $app->terminate('in callback');
});
```

3.7.3 Callback Triggering

To illustrate how callbacks work, let's imagine the following workflow:

- your application with the above code resides in file 'test.php'
- when user opens 'test.php' in the browser, first 4 lines of code execute but the set() will not execute "terminate". Execution will continue as normal.
- getURL() will provide link e.g. *test.php?app_callback=callback*

When page renders, the user can click on a label. If they do, the browser will send another request to the server:

- this time same request is sent but with the *?app_callback=callback* parameter
- the *`Callback::set()`* will notice this argument and execute "terminate()"
- terminate() will exit app execution and output 'in callback' back to user.

Calling *`App::terminate()`* will prevent the default behaviour (of rendering UI) and will output specified string instead, stopping further execution of your application.

3.7.4 Return value of set()

The callback verifies trigger condition when you call *`Callback::set()`*. If your callback returns any value, the set() will return it too:

```
$label = $app->add(['Label','Callback URL:']);
$cb = $label->add('Callback');
$label->detail = $cb->getURL();
$label->link($cb->getURL());

if($cb->set(function(){ return true; })) {
    $label->addClass('red');
}
```

This example uses return of the *`Callback::set()`* to add class to a label, however a much more preferred way is to use `$triggered`.

property `Callback::$`**triggered**

You use property *triggered* to detect if callback was executed or not, without short-circuting the execution with set() and terminate(). This can be helpful sometimes when you need to affect the rendering of the page through a special call-back link. The next example will change color of the label regardless of the callback function:

```
$label = $app->add(['Label','Callback URL:']);
$cb = $label->add('Callback');
$label->detail = $cb->getURL();
$label->link($cb->getURL());

$cb->set(function(){ echo 123; });

if ($cb->triggered) {
    $label->addClass('red');
}
```

property Callback::$**POST_trigger**

A Callback class can also use a POST variable for triggering. For this case the $callback->name should be set through the POST data.

Even though the functionality of Callback is very basic, it gives a very solid foundation for number of derived classes.

3.7.5 CallbackLater

class CallbackLater

This class is very similar to Callback, but it will not execute immediatelly. Instead it will be executed either at the end at beforeRender or beforeOutput hook from inside App, whichever comes first.

In other words this won't break the flow of your code logic, it simply won't render it. In the next example the $label->detail is assigned at the very end, yet callback is able to access the property:

```
$label = $app->add(['Label','Callback URL:']);
$cb = $label->add('CallbackLater');

$cb->set(function() use($app, $label) {
    $app->terminate('Label detail is '.$label->detail);
});

$label->detail = $cb->getURL();
$label->link($cb->getURL());
```

CallbackLater is used by several actions in Agile UI, such as jsReload(), and ensures that the component you are reloading are fully rendered by the time callback is executed.

Given our knowledge of Callbacks, lets take a closer look at how jsReload actually works. So what do we know about *jsReload* already?

- jsReload is class implementing jsExpressionable
- you must specify a view to jsReload
- when triggered, the view will refresh itself on the screen.

Here is example of jsReload:

```
$view = $app->add(['ui'=>'tertiary green inverted segment']);
$button = $app->add(['Button', 'Reload Lorem']);

$button->on('click', new \atk4\ui\jsReload($view));

$view->add('LoremIpsum');
```

NOTE: that we can't perform jsReload on LoremIpsum directly, because it's a text, it needs to be inside a container. When jsReload is created, it transparently creates a 'CallbackLater' object inside *$view*. On the JavaScript side, it will execute this new route which will respond with a NEW content for the $view object.

Should jsReload use regular 'Callback', then it wouldn't know that $view must contain LoremIpsum text.

jsReload existance is only possible thanks to CallbackLater implementation.

3.7.6 jsCallback

class jsCallback

So far, the return value of callback handler was pretty much insignificant. But wouldn't it be great if this value was meaningful in some way?

jsCallback implements exactly that. When you specify a handler for jsCallback, it can return one or multiple *Actions* which will be rendered into JavaScript in response to triggering callback's URL. Let's bring up our older example, but will use jsCallback class now:

```
$label = $app->add(['Label','Callback URL:']);
$cb = $label->add('jsCallback');

$cb->set(function() {
    return 'ok';
});

$label->detail = $cb->getURL();
$label->link($cb->getURL());
```

When you trigger callback, you'll see the output:

```
{"success":true,"message":"Success","eval":"alert(\"ok\")"}
```

This is how jsCallback renders actions and sends them back to the browser. In order to retrieve and execute actions, you'll need a JavaScript routine. Luckily jsCallback also implements jsExpressionable, so it, in itself is an action.

Let me try this again. jsCallback is an *Actions* which will execute request towards a callback-URL that will execute PHP method returning one or more *Actions* which will be received and executed by the original action.

To fully use jsAction above, here is a modified code:

```
$label = $app->add(['Label','Callback URL:']);
$cb = $label->add('jsCallback');

$cb->set(function() {
    return 'ok';
});

$label->detail = $cb->getURL();
$label->on('click', $cb);
```

Now, that is pretty long. For your convenience, there is a shorter mechanism:

```
$label = $app->add(['Label', 'Callback test']);

$label->on('click', function() {
    return 'ok';
});
```

3.7.6.1 User Confirmation

The implementation perfectly hides existence of callback route, javascript action and jsCallback. The jsCallback is based on 'Callback' therefore code after `View::on()` will not be executed during triggering.

property `jsCallback::$`**`confirm`**

If you set *confirm* property action will ask for user's confirmation before sending a callback:

```
$label = $app->add(['Label','Callback URL:']);
$cb = $label->add('jsCallback');

$cb->confirm = 'sure?';

$cb->set(function() {
    return 'ok';
});

$label->detail = $cb->getURL();
$label->on('click', $cb);
```

This is used with delete operations. When using `View::on()` you can pass extra argument to set the 'confirm' property:

```
$label = $app->add(['Label', 'Callback test']);

$label->on('click', function() {
    return 'ok';
}, ['confirm'=>'sure?']);
```

3.7.6.2 JavaScript arguments

`jsCallback::`**`set`**(*$callback*, *$arguments* =[ ])

It is possible to modify expression of jsCallback to pass additional arguments to it's callback. The next example will send browser screen width back to the callback:

```
$label = $app->add('Label');
$cb = $label->add('jsCallback');

$cb->set(function($j, $arg1){
    return 'width is '.$arg1;
}, [new \atk4\ui\jsExpression( '$(window).width()' )]);

$label->detail = $cb->getURL();
$label->js('click', $cb);
```

In here you see that I'm using a 2nd argument to $cb->set() to specify arguments, which, I'd like to fetch from the browser. Those arguments are passed to the callback and eventually arrive as $arg1 inside my callback. The `View::on()` also supports argument passing:

```
$label = $app->add(['Label', 'Callback test']);

$label->on('click', function($j, $arg1) {
    return 'width is '.$arg1;
}, ['confirm'=>'sure?', 'args'=>[new \atk4\ui\jsExpression( '$(window).width()' )]]);
```

If you do not need to specify confirm, you can actually pass arguments in a key-less array too:

```
$label = $app->add(['Label', 'Callback test']);

$label->on('click', function($j, $arg1) {
    return 'width is '.$arg1;
}, [new \atk4\ui\jsExpression( '$(window).width()' )]);
```

3.7.6.3 Refering to event origin

You might have noticed that jsCallback now passes first argument ($j) which so far, we have ignored. This argument is a jQuery chain for the element which received the event. We can change the response to do something with this element. Instead of *return* use:

```
$j->text('width is '.$arg1);
```

Now instead of showing an alert box, label content will be changed to display window width.

There are many other applications for jsCallback, for example, it's used in `Form::onSubmit()`.

3.8 VirtualPage

Building on the foundation of *Callbacks*, components `VirtualPage` and `Loader` exist to enhance other Components with dynamically loadable content. Here is example for `Tabs`:

```
$tabs = $app->add('Tabs');
$tabs->addTab('First tab is static')->add('LoremIpsum');

$tabs->addTab('Second tab is dynamic', function($vp) {
    $vp->add('LoremIpsum');
});
```

As you switch between those two tabs, you'll notice that the `Button` label on the "Second tab" reloads every time. `Tabs` implements this by using `VirtualPage`, read further to find out how:

3.8.1 VirtualPage Introduction

Before learning about VirtualPage, Loader and other ways of dynamic content loading, you should fully understand *Callbacks*.

`class VirtualPage`

Unlike any of the Callback classes, VirtualPage is a legit `View`, but it's behaviour is a little "different". In normal circumstances, rendering VirtualPage will result in empty string. Adding VirtualPage anywhere inside your *Render Tree* simply won't have any visible effect:

```
$vp = $layout->add('VirtualPage');
$vp->add('LoremIpsum');
```

However, VirtualPage has a special trigger argument. If found, then VirtualPage will interrupt normal rendering progress and output HTML of itself and any other Components you added to that page.

To help you understand when to use VirtualPage here is the example:

- Create a `Button`

- Add VirtualPage inside a button.
- Add Form inside VirtualPage.
- Clicking the Button would dynamically load contents of VirtualPage inside a Modal window.

This pattern is very easy to implement and is used by many components to transparently provide dynamic functionality. Next is an example where `Tabs` has support for call-back for generating dynamic content for the tab:

```
$tabs->addTab('Dynamic Tab Content', function($vp) {
    $vp->add('LoremIpsum');
});
```

Using VirtualPage inside your component can significantly enhance usability without introducing any complexity for developers.

(For situations when Component does not natively support VirtualPage, you can still use *Loader*, documented below).

property `VirtualPage::$`**`$cb`**

VirtuaPage relies on *CallbackLater* object, which is stored in a property $cb. If the Calllback is triggered through a GET argument, then VirtualPage will change it's rendering technique. Lets examine it in more detail:

```
$vp = $layout->add('VirtualPage');
$vp->add('LoremIpsum');

$label = $layout->add('Label');

$label->detail = $vp->cb->getURL();
$label->link($vp->cb->getURL());
```

This code will only show the link containing a URL, but will not show LoremIpsum text. If you do follow the link, you'll see only the 'LoremIpsum' text.

3.8.1.1 Output Modes

`VirtualPage::`**`getURL`**(*$mode = 'callback'*)

VirtualPage can be used to provide you either with RAW HTML content or wrap it into boilerplate HTML. As you may know, *Callback::getURL()* accepts an argument, and VirtualPage gives this argument meaning:

- getURL('cut') gives you URL which will return ONLY the HTML of virtual page, no Layout or boilerplate.
- getURL('popup') gives you URL which will return a very minimalistic layout inside a valid HTML boilerplate, suitable for iframes or popup windows.

You can experement with:

```
$label->detail = $vp->cb->getURL('popup');
$label->link($vp->cb->getURL('popup'));
```

3.8.1.2 Setting Callback

`VirtualPage::`**`set`**(*$callback*)

Although VirtualPage can work without defining a callback, using one is more reliable and is always recommended:

```
$vp = $layout->add('VirtualPage');
$vp->set(function($vp){
    $vp->add('LoremIpsum');
});

$label = $layout->add('Label');

$label->detail = $vp->cb->getURL();
$label->link($vp->cb->getURL());
```

This code will perform identically as the previous example, however 'LoremIpsum' will never be initialized unless you are requesting VirtualPage specifically, saving some CPU time. Capability of defining callback also makes it possible for VirtualPage to be embedded into any *Component* quite reliably.

To illustrate, see how `Tabs` component rely on VirtualPage, the following code:

```
$t = $layout->add('Tabs');

$t->addTab('Tab1')->add('LoremIpsum'); // regular tab
$t->addTab('Tab2', function($p){ $p->add('LoremIpsum'); }); // dynamic tab
```

`VirtualPage::`**`getURL`**`(`*$html_wrapping*`)`

You can use this shortcut method instead of $vp->cb->getURL().

`property` `VirtualPage::$`**`$ui`**

When using 'popup' mode, the output appears inside a *<div class="ui container">*. If you want to change this class, you can set $ui property to something else. Try:

```
$vp = $layout->add('VirtualPage');
$vp->add('LoremIpsum');
$vp->ui = 'red inverted segment';

$label = $layout->add('Label');

$label->detail = $vp->cb->getURL('popup');
$label->link($vp->cb->getURL('popup'));
```

3.8.2 Loader

`class Loader`

`Loader::`**`set`**`()`

Loader is designed to delay some slow-loading content by loading it dynamically, after main page is rendered.

Comparing to VirtualPage which is a D.Y.I. solution - Loader can be used out of the box. Loader extends VirtualPage and is quite similar to it.

Like with a VirtualPage - you should use *set()* to define content that will be loaded dynamically, while a spinner is shown to a user:

```
$loader = $app->add('Loader');
$loader->set(function($p) {

    // Simulate slow-loading component
    sleep(2);
    $p->add('LoremIpsum');
});
```

A good use-case example would be a dashboard graph. Unlike VirtualPage which is not visible to a regular render, Loader needs to occupy some space.

property Loader::$**$shim**

By default it will display a white segment with 7em height, but you can specify any other view thorugh $shim property:

```
$loader = $app->add(['Loader', 'shim'=>['Message', 'Please wait until we load␣
↪LoremIpsum...', 'red']]);
$loader->set(function($p) {

    // Simulate slow-loading component
    sleep(2);
    $p->add('LoremIpsum');

});
```

3.8.2.1 Triggering Loader

By default, Loader will display a spinner and will start loading it's contents as soon as DOM Ready() event fires. Sometimes you want to control the event.

Loader::**jsLoad**(*$args* =[ ])

Returns JS action which will trigger loading. The action will be carried out in 2 steps:

- loading indicator will be displayed
- JS will request content from $this->getURL() and provided by set()
- Content will be placed inside Loader's DIV replacing shiv (or previously loaded content)
- loading indicator will is hidden

$loadEvent = null

If you have NOT invoked jsLoad in your code, Loader will automatically assign it do DOM Ready(). If the automatic behaviour does not work, you should set value for $loadEvent:

- null = load on DOM ready unless you have invoked jsLoad() in the code.
- true = load on DOM ready
- false = never load
- "string" - bind to custom JS event

To indicate how custom binding works:

```
$loader = $app->add(['Loader', 'loadEvent' => 'kaboom']);

$loader->set(function($p){
    $p->add('LoremIpsum');
});


$app->add(['Button', 'Load data'])->on('click', $loader->js()->trigger('kaboom'));
```

This approach allow you to trigger loader from inside JavaScript easily. See also: http://api.jquery.com/trigger/

3.8.2.2 Reloading

If you execute *jsReload* action on the Loader, it will return to original state.

3.8.2.3 Inline Editing Example

Next example will display DataTable, but will allow you to repalce data with a form temporarily:

```
$box = $app->add(['ui'=>'segment']);

$loader = $box->add(['Loader', 'loadEvent'=>'edit']);
$loader->add('Table')
    ->setModel($data)
    ->addCondition('year', $app->stickyGet('year'));

$box->add(['Button', 'Edit Data Settings'])->on('click', $loader->js()->trigger('edit
↪'));

$loader->set(function($p) use($loader) {
    $form = $p->add('Form');
    $form->addField('year');

    $form->onSubmit(function($form) use ($loader) {
        return new \atk4\ui\jsReload($loader, ['year'=>$form->model['year']]);
    });
});
```

3.8.2.4 Progress Bar

$progressBar = null

Loader can have a progress bar. Imagine that your Loader has to run slow process 4 times:

```
sleep(1);
sleep(1);
sleep(1);
sleep(1);
```

You can notify user about this progress through a simple code:

```
$loader = $app->add(['Loader', 'progressBar'=>true]);
$loader->set(function($p) {

    // Simulate slow-loading component
    sleep(1);
    $p->setProgress(0.25);
    sleep(1);
    $p->setProgress(0.5);
    sleep(1);
    $p->setProgress(0.75);
    sleep(1);

    $p->add('LoremIpsum');

});
```

By setting progressBar to true, Loader component will use SSE (Server Sent Events) and will be sending notification about your progress. Note that currently Internet Explorer does not support SSE and it's up to you to create a work-around.

Agile UI will test your browser and if SSE are not supported, $progressBar will be ignored.

3.9 Documentation is coming soon.

CHAPTER 4

Components

Classes that extend from `View` are called *Components* and inherit abilities to render themselves (see *Render Tree*)

4.1 Core Components

Some components serve as a foundation of entire set of other components. A lot of qualities implemented by a core component is inherited by its descendants.

4.1.1 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.

4.1.1.1 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 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']);
$top = new \atk4\ui\View(['ui'=>'segments']);
$bottom = new \atk4\ui\Button(['Hello World', 'orange']);

// not arranged into render-tree yet

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


// Still not sure if finished adding

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

// 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');
$top = $app->setLayout(new \atk4\ui\View(['ui'=>'segments']));

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

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

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

```
$app = new \atk4\ui\App('My App');
$top = $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)

4.1.1.2 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.

4.1.1.3 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'.

4.1.1.4 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 varietty 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 = $app->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 = $app->add('View', ['ui'=>'segment', 'id'=>'test-id']);
```

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

```
$view = $app->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 = $app->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 clases 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"

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

This code will have same effect as:

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

4.1.1.6 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.

4.1.1.7 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:

```
$app->add(new MyView(), ['template'=>$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 = $app->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`*.

4.1.1.8 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 derrived 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 througout 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' propoerty.

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
```

4.1.1.9 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', '
```

4.1.1.10 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::**setProperties**(*$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**()

4.1.2 Lister

class atk4\ui\Lister

Lister can be used to output unstructured data with your own HTML template. If you wish to output data in a table, see *Table*. Lister is also the fastest way to render large amount of output and will probably give you most flexibility.

4.1.2.1 Basic Usage

The most common use is when you need to implement a certain HTML and if that HTML contains list of items. If your HTML looks like this:

```
<div class="ui header">Top 20 countries (alphabetically)</div>
  <div class="ui icon label"><i class="ae flag"></i> Andorra</div>
  <div class="ui icon label"><i class="cm flag"></i> Camerroon</div>
  <div class="ui icon label"><i class="ca flag"></i> Canada</div>
</div>
```

you should put that into file *myview.html* then use it with a view:

```
$view = $app->add(['template'=>'myview.html']);
```

Now your application should contain list of 3 sample countires as you have specified in HTML, but next we need to add some tags into your template:

```
<div class="ui header">Top {limit}20{/limit} countries (alphabetically)</div>
  {Countries}
  {rows}
  {row}
  <div class="ui icon label"><i class="ae flag"></i> Andorra</div>
  {/row}
  <div class="ui icon label"><i class="cm flag"></i> Camerroon</div>
  <div class="ui icon label"><i class="ca flag"></i> Canada</div>
  {/rows}
  {/Countries}
</div>
```

Here the *{Countries}* region will be replaced with the lister, but the contents of this region will be re-used as the list template. Refresh your page and your output should not be affected at all, becuse View clears out all extra template tags.

Next I'll add Lister:

```
$view->add('Lister', 'Countries')
    ->setModel(new Country($db))
    ->setLimit(20);
```

While most other objects in Agile UI come with their own templates, lister will prefer to use template inside your region. It will look for "row" and "rows" tag:

1. Create clone of {row} tag
2. Delete contents of {rows} tag
3. For each model row, populate values into {row}
4. Render {row} and append into {rows}

If you refresh your page now, you should see "Andorra" duplicated 20 times. This is because the {row} did not contain any field tags. Lets set them up:

```
{row}
<div class="ui icon label"><i class="{iso}ae{/} flag"></i> {name}Andorra{/name}</div>
{/row}
```

Refresh your page and you should see list of countries as expected. The flags are not showing yet, but I'll deal with in next section. For now, lets clean up the template by removing unnecessary tag content:

```
<div class="ui header">Top {limit}20{/limit} countries (alphabetically)</div>
  {Countries}
  {rows}
  {row}
  <div class="ui icon label"><i class="{$iso} flag"></i> {$name}</div>
  {/row}
  {/rows}
  {/Countries}
</div>
```

Finally, Lister permits you not to use {rows} and {row} tags if entire region can be considered as a row:

```
<div class="ui header">Top {limit}20{/limit} countries (alphabetically)</div>
  {Countries}
  <div class="ui icon label"><i class="{$iso} flag"></i> {$name}</div>
  {/Countries}
</div>
```

4.1.2.2 Tweaking the output

Output is formatted using the standard ui_persistence routine, but you can also fine-tune the content of your tags like this:

```
$lister->addHook('beforeRow', function($l){
    $l->current_row['iso']=strtolower($l->current_row['iso']);
})
```

4.1.2.3 Model vs Static Source

Since Lister is non-interractive, you can also set a static source for your lister to avoid hassle:

```
$lister->setSource([
    ['flag'=>'ca', 'name'=>'Canada'],
    ['flag'=>'uk', 'name'=>'UK'],
]);
```

4.1.2.4 Special template tags

Your {row} tempalte may contain few special tags:

- {$_id} - will be set to ID of the record (regardless of how your id field is called)
- {$_title} - will be set to the title of your record (see $model->$title_field)
- {$_href} - will point to current page but with ?id=123 extra GET argument.

4.1.2.5 Using without Template

Agile UI comes with a one sample template for your lister, although it's not set by default, you can specify it explicitly:

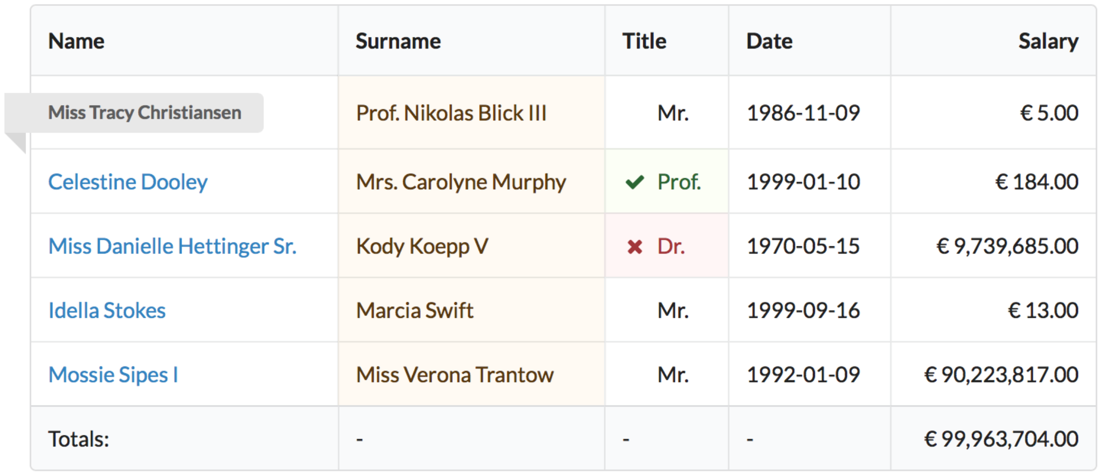
```
$app->add(['Lister', 'defaultTemplate'=>'lister.html']);
```

This should display a list nicely formatted by Semantic UI, with header, links, icons and description area.

4.1.3 Table

class atk4\ui\Table

Table is the simplest way to output multiple records of structured, static data. For Un-structure output please see *Lister*



Name	Surname	Title	Date	Salary
Miss Tracy Christiansen	Prof. Nikolas Blick III	Mr.	1986-11-09	€ 5.00
Celestine Dooley	Mrs. Carolyne Murphy	✓ Prof.	1999-01-10	€ 184.00
Miss Danielle Hettinger Sr.	Kody Koepp V	✖ Dr.	1970-05-15	€ 9,739,685.00
Idella Stokes	Marcia Swift	Mr.	1999-09-16	€ 13.00
Mossie Sipes I	Miss Verona Trantow	Mr.	1992-01-09	€ 90,223,817.00
Totals:	-	-	-	€ 99,963,704.00

Various composite components use Table as a building block, see *Grid* and CRUD. Main features of Table class are:

- Tabular rendering using column headers on top of markup of https://semantic-ui.com/collections/table.html.
- Support for data formatting. (money, dates, etc)
- Column decorators, icons, buttons, links and color.
- Support for "Totals" row.
- Can use Agile Data source or Static data.
- Custom HTML, Format hooks

4.1.3.1 Basic Usage

The simplest way to create a table is when you use it with Agile Data model:

```
$table = $app->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.

Using with Array Data

You can also use Table with Array data source like this:

```
$my_array = [
    ['name'=>'Vinny', 'surname'=>'Sihra', 'birthdate'=>new \DateTime('1973-02-03')],
    ['name'=>'Zoe', 'surname'=>'Shatwell', 'birthdate'=>new \DateTime('1958-08-21')],
    ['name'=>'Darcy', 'surname'=>'Wild', 'birthdate'=>new \DateTime('1968-11-01')],
    ['name'=>'Brett', 'surname'=>'Bird', 'birthdate'=>new \DateTime('1988-12-20')],
];

$table = $app->add('Table');
$table->setSource($my_array);

$table->addColumn('name');
$table->addColumn('surname', ['Link', 'url'=>'details.php?surname={$surname}']);
$table->addColumn('birthdate', null, ['type'=>'date']);
```

Warning: I encourage you to seek appropriate Agile Data persistence instead of handling data like this. The implementation of `View::setSource` will create a model for you with Array persistence for you anyways.

Adding Columns

atk4\ui\Table::**setModel**(*atk4dataModel $m*, *$fields = null*)

atk4\ui\Table::**addColumn**(*$name*, *$columnDecorator = null*, *$field = null*)

To change the order or explicitly specify which field columns must appear, if you pass list of those fields as second argument to setModel:

```
$table = $app->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.

You can also add individual column to your table:

```
$table->setModel(new Order($db), false); // false here means - don't add any fields␣
↪by default
$table->addColumn('name');
$table->addColumn('price');
```

When invoking addColumn, you have a great control over the field properties and decoration. The format of addColumn() is very similar to `Form::addField`.

4.1.3.2 Calculations

Apart from adding columns that reflect currrent values of your database, there are several ways how you can calculate additional values. You must know the capabilities of your database server if you want to execute some calculation there. (See http://agile-data.readthedocs.io/en/develop/expressions.html)

It's always a good idea to calculate column inside datababase. Lets create "total" column which will multiply "price" and "amount" values. Use `addExpression` to provide in-line definition for this field if it's not alrady defined in `Order::init()`:

```
$table = $app->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 = $app->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

To read more about column objects, see tablecolumn

Advanced Column Denifitions

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 `TableColumn\Generic` then *`Table`* will respord 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.

addColumn adds a new column to the table. This method was explained above but can also be used to add colums without field:

```
$action = $this->addColumn(null, ['Actions']);
$actions->addAction('Test', function() { return 'ok'; });
```

The above code will add a new extra column that will only contain 'delete' icon. When clicked it will automatically delete the corresponding 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 not be associated with any model field in `TableColumnGeneric::getHeaderCellHTML`, `TableColumnGeneric::getTotalsCellHTML` and `TableColumnGeneric::getDataCellHTML`.

Some columns require name, such as *`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 = $app->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 = $app->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.

addColumns($names);

Here, names can be an array of strings ([‘status’, ‘price’]) or contain array that will be passed as argument sto 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->addColumns(['total','status'])
$table->addColumn($c_gap);
```

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

4.1.3.3 Table sorting

property atk4\ui\Table::$**sortable**

property atk4\ui\Table::$**sort_by**

property atk4\ui\Table::$**sort_order**

Table does not support an interractive sorting on it's own, (but *Grid* does), however you can designade 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).

4.1.3.4 Talbe Data Handling

Table is very similar to *Lister* 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);

$app->add('Table')->setModel($files);


// Show contents of dropbox
$github = \atk4\github\Persistence_Issues($github_api_config);
$issues = new \atk4\github\Model\Issue($github);

$app->add('Table')->setModel($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 persitences. (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.

4.1.3.5 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. Formattrers are always applied in the same orders they are defined. Remember that setModel() will typically set a Generic fromatter 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="{$c_name_link}">£ {$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}">£ {$salary}</a></td>
```

Which is concatinated 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 concatinated 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 geterating 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 overriden getDataCellHTML only, then your column will still work OK provided that it's used as a last formatter.

4.1.3.6 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*

4.1.3.7 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 contens 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`*.

4.1.3.8 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 furteher)

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 asign 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 interract with each row directly.

The basic usage format is:

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

4.1.4 Form Field Decorators

Agile UI dedicates a separate namespace for the Form Field Decorator. Those are quite simple components that present themselves as input controls: line, select, checkbox.

4.1.4.1 Relationship with Form

All Field Decorators can be integrated with `atk4uiForm` which will facilitate collection and processing of data in a form. Field decorators can also be used as stand-alone controls.

Stand-alone use

atk4\ui\FormField\set()

atk4\ui\FormField\jsInput()

Add any field decorator to your application like this:

```
$field = $app->add(new \atk4\ui\FormField\Line());
```

You can set default value and ineract with a field using JavaScript:

```
$field->set('hello world');


$button = $app->add(['Button', 'check value']);
$button->on('click', new \atk4\ui\jsExpression('alert("field value is: "+[])', [
↪$field->jsInput()->val()]));
```

When used stand-alone, FormFields will produce a basic HTML (I have omitted id=):

```
<div class="ui  input">
    <input name="line" type="text" placeholder="" value="hello world"/>
</div>
```

Using in-form

Field can also be used inside a form like this:

```
$form = $app->add('Form');
$field = $form->addField('name', new \atk4\ui\FormField\Line());
```

If you execute this exmple, you'll notice that Feld now has a label, it uses full width of the page and the following HTML is now produced:

```
<div class="field">
  <label for="atk_admin_form_generic_name_input">Name</label>
  <div id="atk_admin_form_generic_name" class="ui input" style="">
    <input name="name" type="text" placeholder="" id="atk_admin_form_generic_name_
↪input" value="">
  </div>
</div>
```

The markup that surronds the button which includes Label and formatting is produced by `atk4uiFormLayoutGeneric`, which does draw some of the information from the Field itself.

Using in Form Layouts

Form may have multiple Form Layouts and that's very useful if you need to split up form into multiple Tabs or detach field groups or even create nested layouts:

```
$form = $app->add('Form');
$tabs = $form->add('Tabs', 'AboveFields');
$form->add(['ui'=>'divider'], 'AboveFields');

$form_page = $tabs->addTab('Basic Info')->add(['FormLayout\Generic', 'form'=>$form]);
$form_page->addField('name', new \atk4\ui\FormField\Line());

$form_page = $tabs->addTab('Other Info')->add(['FormLayout\Generic', 'form'=>$form]);
$form_page->addField('age', new \atk4\ui\FormField\Line());

$form->onSubmit(function($f) {  return $f->model['name'].' has age '.$f->model['age'];
↪ });
```

This is further explained in documentation for `atk4uiFormLayoutGeneric` class, however if you do plan on adding your own field types, it's important that you extend it properly:

- Generic (abstract, extends View) - Use this if field is NOT based on *<input>*
- Input (abstract, extends Generic) - Easiest since it alrady implements *<input>* and various ways to attach button to the input with markup of Semantic UI field.

4.1.4.2 Relatioship with Model

In the examples above, we looked at the manual use where you create Field Decorator object explicitly. The most common use-case in large application is use with Models. You would need a model, such as *Country* model (see demos/database.php) as well as Persistence $db.

Now, in order to create a form, the following is sufficient:

```
$form = $app->add('Form');
$form->setModel(new Country($db);
```

The above will populate fields from model into the form automatically. You can use second argument to `atk4uiForm::setModel()` to indicate which fields to display or rely on field_visibility.

When Form fields are populated, then `atk4uiForm::_decoratorFactory` is consulted to make a decision on how to translate Model Field into Form Field Decorator.

The rules are rather straightforward but may change in future versions of Agile UI:

- if enum is defined, use *DropDown*
- consult `atk4uiForm::$typeToDecorator` property for type-to-seed association
- type=password will use `Password`

You always have an option to explicitly specify which field you would like to use:

```
$model->addField('long_text', ['ui'=>['Form'=>'TextArea']]);
```

It is recommended however, that you use type when possible, because types will be universally supported by all components:

```
$model->addField('long_text', ['type'=>'text']);
```

Note: All forms will be associted with a model. If form is not explicitly linked with a model, it will create a ProxyModel and all fields will be created automatically in that model. As a result, all Field Decorators will be linked with Model Fields.

Link to Model Field

property atk4\ui\FormField\**$field**

Form decorator defines $field property which will be pointing to a field object of a model, so technically the value of the field would be read from *$decorator->field->get()*.

4.1.4.3 Line Input Field

class atk4\ui\FormField\Input

Implements View for presenting Input fields. Based around http://semantic-ui.com/elements/input.html.

Similar to other views, Input has various properties that you can specify directly or inject through constructor. Those properties will affect the look of the input element. For example, *icon* property:

Here are few ways to specify *icon* to an Input:

```
// compact
$page->add(new \atk4\ui\FormField\Line('icon'=>'search'));

// Type-hinting friendly
$line = new \atk4\ui\FormField\Line();
$line->icon='search';
$page->add($line);

// using class factory
$page->add('FormField/Line', ['icon'=>'search']);
```

The 'icon' property can be either string or a View. The string is for convenience and will be automatically substituted with *new Icon($icon)*. If you wish to be more specifc and pass some arguments to the icon, there are two options:

```
// compact
$line->icon=['search', 'big'];

// Type-hinting friendly
$line->icon = new Icon('search');
$line->icon->addClass('big');
```

To see how Icon interprets *new Icon(['search', 'big'])*, refer to `Icon`.

Note: View's constructor will map received arguments into object properties, if they are defined or addClass() if not. See *View::setProperties*.

property atk4\ui\FormField\Input::$**placeholder**

Will set placeholder property.

property atk4\ui\FormField\Input::$**loading**

Set to "left" or "right" to display spinning loading indicator.

property atk4\ui\FormField\Input::$**label**

property atk4\ui\FormField\Input::$**labelRight**

Convert text into `Label` and insert it into the field.

property atk4\ui\FormField\Input::$**action**

property atk4\ui\FormField\Input::$**actionLeft**

Convert text into `Button` and insert it into the field.

To see various examples of fields and their attributes see *demos/field.php*.

Integration with Form

When you use `form::addField()` it will create 'Field Decorator'

JavaScript on Input

`atk4\ui\FormField\Input::`**`jsInput`**`(`[*$event*[, *$other_action*]])

Input class implements method jsInput which is identical to `View::js`, except that it would target the INPUT element rather then the whole field:

```
$field->jsInput(true)->val(123);
```

4.1.4.4 DropDown

class `atk4\ui\FormField\`**`DropDown`**

property `atk4\ui\FormField\DropDown::$`**`$values`**

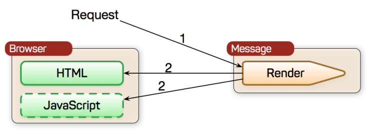
property `atk4\ui\FormField\DropDown::$`**`$empty`**

4.1.4.5 AutoComplete

class `atk4\ui\FormField\`**`AutoComplete`**

4.2 Simple components

Simple components exist for the purpose of abstraction and creating a decent interface which you can rely on when programming your PHP application with Agile UI. In some cases it may make sense to rely on HTML templates for the simple elements such as Icons, but when you are working with dynamic and generic components quite often you need to abstract HTML yet let the user have decent control over even the small elements.



4.2.1 Button

class `atk4\ui\`**`Button`**

Implements a clickable button:

```
$button = $app->add(['Button', 'Click me']);
```

The Button will typically inherit all same properties of a `View`. The base class "View" implements many useful methods already, such as:

```
$button->addClass('big red');
```

Alternatvie syntax if you wish to initialize object yourself:

```
$button = new Button('Click me');
$button->addClass('big red');

$app->add($button);
```

You can refer to the Semantic UI documentation for Button to find out more about available classes: http://semantic-ui.com/elements/button.html.

Demo: http://ui.agiletoolkit.org/demos/button.php

4.2.1.1 Button Icon

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

Property $icon will place icon on your button and can be specified in one of the following two ways:

```
$button = $app->add(['Button', 'Like', 'blue', 'icon'=>'thumbs up']);

// or

$button = $app->add(['Button', 'Like', 'blue', 'icon'=>new Icon('thumbs up')]);
```

or if you prefer initializing objects:

```
$button = new Button('Like');
$button->addClass('blue');
$button->icon = new Icon('thumbs u');

$app->add($button);
```

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

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

```
$button = $app->add(['Button', 'Next', 'iconRight'=>'right arrow']);
```

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

4.2.1.2 Button Bar

Buttons can be aranged into a bar. You would need to create a *View* component with property `ui='buttons'` and add your other buttons inside:

```
$bar = $app->add(['ui'=>'vertical buttons']);

$bar->add(['Button', 'Play', 'icon'=>'play']);
$bar->add(['Button', 'Pause', 'icon'=>'pause']);
$bar->add(['Button', 'Shuffle', 'icon'=>'shuffle']);
```

At this point using alternative syntax where you initialize objects yourself becomes a bit too complex and lengthy:

```
$bar = new View();
$bar->ui = 'buttons';
$bar->addClass('vertical');

$button = new Button('Play');
$button->icon = 'play';
$bar->add($button);

$button = new Button('Pause');
$button->icon = 'pause';
```

```
$bar->add($button);

$button = new Button('Shuffle');
$button->icon = 'shuffle';
$bar->add($button);

$app->add($bar);
```

4.2.1.3 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()'));
```

4.2.1.4 Complex Buttons

Knowledge of the 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']));
$app->add($forks);
```

4.2.2 Label

class `atk4\ui\`**`Label`**

Labels can be used in many different cases, either as a stand-alone objects, inside tables or inside other components.

To see what possible classes you can use on the Label, see: http://semantic-ui.com/elements/label.html.

Demo: http://ui.agiletoolkit.org/demos/label.php

4.2.2.1 Basic Usage

First argument of constructor or first element in array passed to constructor will be the text that will appear on the label:

```
$label = $app->add(['Label', 'hello world']);

// or
```

```
$label = new \atk4\ui\Label('hello world');
$app->add($label);
```

Label has the following properties:

property atk4\ui\Label::$**icon**

property atk4\ui\Label::$**iconRight**

property atk4\ui\Label::$**image**

property atk4\ui\Label::$**imageRight**

property atk4\ui\Label::$**detail**

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

4.2.2.2 Icons

There are two properties (icon, iconRight) but you can set only one at a time:

```
$app->add(['Label', '23', 'icon'=>'mail']);
$app->add(['Label', 'new', 'iconRight'=>'delete']);
```

You can also specify icon as an object:

```
$app->add(['Label', 'new', 'iconRight'=>new \atk4\ui\Icon('delete')]);
```

For more information, see: *Icon*

4.2.2.3 Image

Image cannot be specified at the same time with the icon, but you can use PNG/GIF/JPG image on your label:

```
$img = 'https://cdn.rawgit.com/atk4/ui/07208a0af84109f0d6e3553e242720d8aeedb784/
↪public/logo.png';
$app->add(['Label', 'Coded in PHP', 'image'=>$img]);
```

4.2.2.4 Detail

You can specify "detail" component to your label:

```
$app->add(['Label', 'Number of lines', 'detail'=>'33']);
```

4.2.2.5 Groups

Label can be part of the group, but you would need to either use custom HTML template or composition:

```
$group = $app->add(['View', false, 'blue tag', 'ui'=>'labels']);
$group->add(['Label', '$9.99']);
$group->add(['Label', '$19.99', 'red tag']);
$group->add(['Label', '$24.99']);
```

4.2.2.6 Combining classes

Based on Semantic UI documentation, you can add more classes to your labels:

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

$c = $columns->addColumn();
$col = $c->add(['View', 'ui'=>'raised segment']);

// attach label to the top of left column
$col->add(['Label', 'Left Column', 'top attached', 'icon'=>'book']);

// ribbon around left column
$col->add(['Label', 'Lorem', 'red ribbon', 'icon'=>'cut']);

// add some content inside column
$col->add(['LoremIpsum', 'size'=>1]);

$c = $columns->addColumn();
$col = $c->add(['View', 'ui'=>'raised segment']);

// attach label to the top of right column
$col->add(['Label', 'Right Column', 'top attached', 'icon'=>'book']);

// some content
$col->add(['LoremIpsum', 'size'=>1]);

// right bottom corner label
$col->add(['Label', 'Ipsum', 'orange bottom right attached', 'icon'=>'cut']);
```

4.2.2.7 Added labels into Table

You can even use label inside a table, but because table renders itself by repeating periodically, then the following code is needed:

```
$table->addHook('getHTMLTags', function ($table, $row) {
    if ($row->id == 1) {
        return [
            'name'=> $table->app->getTag('div', ['class'=>'ui ribbon label'], $row[
↪'name']),
        ];
    }
});
```

Now while $table will be rendered, if it finds a record with id=1, it will replace $name value with a HTML tag. You need to make sure that 'name' column appears first on the left.

4.2.3 Text

class atk4\ui\Text

Text is a component for abstracting several paragraphs of text. It's usage is simple and straightforward:

4.2.3.1 Basic Usage

First argument of constructor or first element in array passed to constructor will be the text that will appear on the label:

```
$text = $app->add(['Text', 'here goes some text']);
```

4.2.3.2 Paragraphs

You can define multiple paragraphs with text like this:

```
$text = $app->add('Text')
    ->addParagraph('First Paragraph')
    ->addParagraph('Second Paragraph');
```

4.2.3.3 HTML escaping

By default Text will not escape HTML so this will render as a bold text:

```
$text = $app->add(['Text', 'here goes <b>some bold text</b>']);
```

Warning: If you are using Text for output HTML then you are doing it wrong. You should use a generic View and specify your HTML as a template.

When you use paragraphs, escaping is performed by default:

```
$text = $app->add('Text')
    ->addParagraph('No alerts')
    ->addParagraph('<script>alert(1)</script>');
```

4.2.3.4 Usage

Text is usable in generic components, where you want to leave possibility of text injection. For instance, `Message` uses text allowing you to add few parapraphs of text:

```
$message = $app->add(['Message', 'Message Title']);
$message->addClass('warning');

$message->text->addParagraph('First para');
$message->text->addParagraph('Second para');
```

4.2.3.5 Limitations

Text may not have embedded elements, although that may change in the future.

4.2.4 LoremIpsum

class atk4\ui\LoremIpsum

This class implements a standard filler-text which is so popular amongst web developers and designers. LoremIpsum will generate a dynamic filler text which should help you test reloading or layouts.

4.2.4.1 Basic Usage

$app->add('LoremIpsum');

4.2.4.2 Resizing

You can define the length of the LoremIpsum text:

```
$text = $app->add('Text')
    ->addParagraph('First Paragraph')
    ->addParagraph('Second Paragraph');
```

You may specify amount of text to be generated with lorem:

```
$app->add(['LoremIpsum', 1]); // just add a little one

// or

$app->add(new LoremIpsum(5)); // adds a lot of text
```

4.2.5 Header

Can be used for page or section headers.

Based around: http://semantic-ui.com/elements/header.html.

Demo: http://ui.agiletoolkit.org/demos/header.php

4.2.5.1 Basic Usage

By default header size will depend on where you add it:

```
$this->add(['Header', 'Hello, Header']);
```

4.2.5.2 Attributes

property atk4\ui\size

property atk4\ui\subHeader

Specify size and sub-header content:

```
$seg->add([
    'Header',
    'H1 header',
    'size'=>1,
    'subHeader'=>'H1 subheader'
]);

// or
```

```
$seg->add([
    'Header',
    'Small header',
    'size'=>'small',
    'subHeader'=>'small subheader'
]);
```

4.2.5.3 Icon and Image

property atk4\ui\icon

property atk4\ui\image

Header may specify icon or image:

```
$seg->add([
    'Header',
    'Header with icon',
    'icon'=>'settings',
    'subHeader'=>'and with sub-header'
]);
```

Here you can also specify seed for the image:

```
$img = 'https://cdn.rawgit.com/atk4/ui/07208a0af84109f0d6e3553e242720d8aeedb784/
↪public/logo.png';
$seg->add([
    'Header',
    'Center-aligned header',
    'aligned'=>'center',
    'image'=>[$img, 'disabled'],
    'subHeader'=>'header with image'
]);
```

4.2.6 Icon

class atk4\ui\Icon

Implements basic icon:

```
$icon = $app->add(new \atk4\ui\Icon('book'));
```

Alternatively:

```
$icon = $app->add('Icon', 'flag')->addClass('outline');
```

Most commonly icon class is used for embedded icons on a `Button` or inside other components (see *Using on other Components*):

```
$b1 = new \atk4\ui\Button(['Click Me', 'icon'=>'book']);
```

You can, of course, create instance of an Icon yourself:

```
$icon = new \atk4\ui\Icon('book');
$b1 = new \atk4\ui\Button(['Click Me', 'icon'=>$icon]);
```

You do not need to add an icon into the render tree when specifying like that. The icon is selected through class. To find out what icons are available, refer to Semantic-UI icon documentation:

https://semantic-ui.com/elements/icon.html

You can also use States, Variations by passing classes to your button:

```
$app->add(new \atk4\ui\Button(['Click Me', 'red', 'icon'=>'flipped big question']));

$app->add(new \atk4\ui\Label(['Battery Low', 'green', 'icon'=>'battery low']));
```

4.2.6.1 Using on other Components

You can use icon on the following components: `Button`, `Label`, `Header` `Message`, `Menu` and possibly some others. Here are some examples:

```
$app->add(new \atk4\ui\Header(['Header', 'red', 'icon'=>'flipped question']));
$app->add(new \atk4\ui\Button(['Button', 'red', 'icon'=>'flipped question']));

$menu = $app->add(new \atk4\ui\Menu());
$menu->addItem(['Menu Item', 'icon'=>'flipped question']);
$sub_menu = $menu->addMenu(['Sub-menu', 'icon'=>'flipped question']);
$sub_menu->addItem(['Sub Item', 'icon'=>'flipped question']);

$app->add(new \atk4\ui\Label(['Label', 'right ribbon red', 'icon'=>'flipped question
↪']));
```

4.2.6.2 Groups

Semantic UI support icon groups. The best way to implement is to supply `Template` to an icon:

```
$app->add(new \atk4\ui\Icon(['template'=>new \atk4\ui\Template('<i class="huge icons">
  <i class="big thin circle icon"></i>
  <i class="user icon"></i>
</i>'), false]));
```

However there are several other options you can use when working with your custom HTML. This is not exclusive to Icon, but I'm adding a few examples here, just for your convenience.

Let's start with a View that contains your custom HTML loaded from file or embedded like this:

```
$view = $app->add(['template'=>new \atk4\ui\Template('<div>Hello my {Icon}<i class=
↪"huge icons">
  <i class="big thin circle icon"></i>
  <i class="{Content}user{/} icon"></i>
</i>{/}, It is me</div>')]);
```

Looking at the template it has a region *{Icon}..{/}*. Try by executing the code above, and you'll see a text message with a user icon in a circle. You can replace this region by passing it as a template into Icon class. For that you need to disable a standard Icon template and specify a correct Spot when adding:

```
$icon = $view->add(new \atk4\ui\Icon(['red book', 'template'=>false]), 'Icon');
```

This technique may be helpful for you if you are creating re-usable elements and you wish to store Icon object in one of your public properties.

Composing

Composing offers you another way to deal with Group icons:

```
$no_users = new \atk4\ui\View([null, 'huge icons', 'element'=>'i']);
$no_users->add(new \atk4\ui\Icon('big red dont'));
$no_users->add(new \atk4\ui\Icon('black user icon'));

$app->add($no_users);
```

4.2.6.3 Icon in Your Component

Sometimes you want to build a component that will contain user-defined icon. Here you can find an implementation for `SocialAdd` component that implements a friendly social button with the following features:

- has a very compact usage `new SocialAdd('facebook')`
- allow to customize icon by specifying it as string, object or injecting properties
- allow to customize label

Here is the code with comments:

```
/**
 * Implements a social network add button. You can initialize the button by passing
 * social network as a parameter: new SocialAdd('facebook')
 * or alternatively you can specify $social, $icon and content individually:
 * new SocialAdd(['Follow on Facebook', 'social'=>'facebook', 'icon'=>'facebook f']);
 *
 * For convenience use this with link(), which will automatically open a new window
 * too.
 */
class SocialAdd extends \atk4\ui\View {
    public $social = null;
    public $icon = null;
    public $defaultTemplate = null;
    // public $defaultTemplate = __DIR__.'../templates/socialadd.html';

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

        if (is_null($this->social)) {
            $this->social = $this->content;
            $this->content = 'Add on '.ucwords($this->content);
        }

        if (!$this->social) {
            throw new Exception('Specify social network to use');
        }

        if (is_null($this->icon)) {
            $this->icon = $this->social;
        }

        if (!$this->template) {
            // TODO: Place template into file and set defaultTemplate instead
            $this->template = new \atk4\ui\Template(
'<{_element}button{/} class="ui '.$this->social.' button" {$attributes}>
  <i class="large icons">
    {$Icon}
    <i class="inverted corner add icon"></i>
  </i>
  {$Content}
</{_element}button{/}>');
        }

        // Initialize icon
        if (!is_object($this->icon)) {
            $this->icon = new \atk4\ui\Icon($this->icon);
        }

        // Add icon into render tree
        $this->add($this->icon, 'Icon');
    }

    function link($url) {
        $this->setAttr('target', '_blank');
        return parent::link($url);
    }
}

// Usage Examples. Start with the most basic usage
$app->add(new SocialAdd('instagram'));

// Next specify label and separatly name of social network
$app->add(new SocialAdd(['Follow on Twitter', 'social'=>'twitter']));

// Finally provide custom icon and make the button clickable.
$app->add(new SocialAdd(['facebook', 'icon'=>'facebook f']))
    ->link('http://facebook.com');
```

4.2.7 Image

class atk4\ui\Image

Implements Image around https://semantic-ui.com/elements/image.html.

4.2.7.1 Basic Usage

Implements basic image:

```
$icon = $app->add(new \atk4\ui\Image('image.gif'));
```

You need to make sure that argumen specified to Image is a valid URL to an image.

4.2.7.2 Specify classes

You can pass additional classes to an image:

```
$img = 'https://cdn.rawgit.com/atk4/ui/07208a0af84109f0d6e3553e242720d8aeedb784/
↪public/logo.png';
$icon = $app->add(['Image', $img, 'disabled']);
```

4.2.8 Message

class atk4\ui\Message

Outputs a rectangular segment with a distinctive color to convey message to the user, based around: https://semantic-ui.com/collections/message.html

Demo: http://ui.agiletoolkit.org/demos/message.php

4.2.8.1 Basic Usage

Implements basic image:

```
$message = new \atk4\ui\Message('Message Title');
$app->add($message);
```

Although typically you would want to specify what type of message is that:

```
$message = new \atk4\ui\Message(['Warning Message Title', 'warning']);
$app->add($message);
```

Here is the alternative syntax:

```
$message = $app->add(['Message', 'Warning Message Title', 'warning']));
```

4.2.8.2 Adding message text

property atk4\ui\Message::$**text**

Property $text is automatically initialized to *Text* so you can call `Text::addParagraph` to add more text inside your message:

```
$message = $app->add(['Message', 'Message Title']);
$message->addClass('warning');
$message->text->addParagraph('First para');
$message->text->addParagraph('Second para');
```

4.2.8.3 Message Icon

property atk4\ui\Message::$**icon**

You can specify icon also:

```
$message = $app->add([
    'Message',
    'Battery low',
    'red',
    'icon'=>'battery low'
])->text->addParagraph('Your battery is getting low. Recharge your Web App');
```

class atk4\ui\Tabs

4.2.9 Tabs

Tabs implement a yet another way to organise your data. The implementation is based on: http://semantic-ui.com/elements/icon.html.

Demo: http://ui.agiletoolkit.org/demos/tabs.php

4.2.9.1 Basic Usage

Once you create Tabs container you can then mix and match static and dynamic tabs:

```
$tabs = $app->add('Tabs');
```

Adding a static conten is pretty simple:

```
$tabs->addTab('Static Tab')->add('LoremIpsum');
```

You can add multiple elements into a single tab, like any other view.

atk4\ui\Tabs::**addTab**(*$name*, *$action*)

Use addTab() method to add more tabs in Tabs view. First parameter is a title of the tab.

Tabs can be static or dynamic. Dynamic tabs use *VirtualPage* implementation mentioned above. You should pass callable action as a second parameter.

Example:

```
$t = $layout->add('Tabs');
```

// add static tab $t->addTab('Static Tab')->add('HelloWorld');

// add dynamic tab $t->addTab('Dynamically Loading', function ($tab) {

$tab->add('LoremIpsum');

});

4.2.9.2 Dynamic Tabs

Dynamic tabs are based around implementation of *VirtualPage* and allow you to pass a call-back which will be triggered when user clicks on the tab.

Note that tab contents are refreshed including any values you put on the form:

```
$t = $app->add('Tabs');

// dynamic tab
$t->addTab('Dynamic Lorem Ipsum', function ($tab) {
    $tab->add(['LoremIpsum', 'size'=>2]);
});

// dynamic tab
$t->addTab('Dynamic Form', function ($tab) {
    $m_register = new \atk4\data\Model(new \atk4\data\Persistence_Array($a));
    $m_register->addField('name', ['caption'=>'Please enter your name (John)']);

    $f = $tab->add(new \atk4\ui\Form(['segment'=>true]));
    $f->setModel($m_register);
    $f->onSubmit(function ($f) {
        if ($f->model['name'] != 'John') {
            return $f->error('name', 'Your name is not John! It is "'.$f->model['name
↪'].'". It should be John. Pleeease!');
        }
    });
});
```

4.2.10 HelloWorld

class atk4\ui\HelloWorld

Presence of the "Hello World" component in the standard distribution is just us saying "The best way to create a Hello World app is around a HelloWorld component".

4.2.10.1 Basic Usage

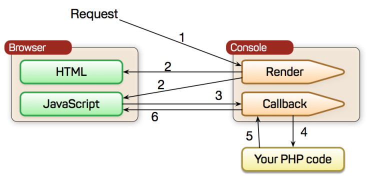
To add a "Hello, World" message:

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

There are no additional features on this component, it is intentionally left simple. For a more sophisticated "Hello, World" implementation look into Hello World add-on.

4.3 Interactive components

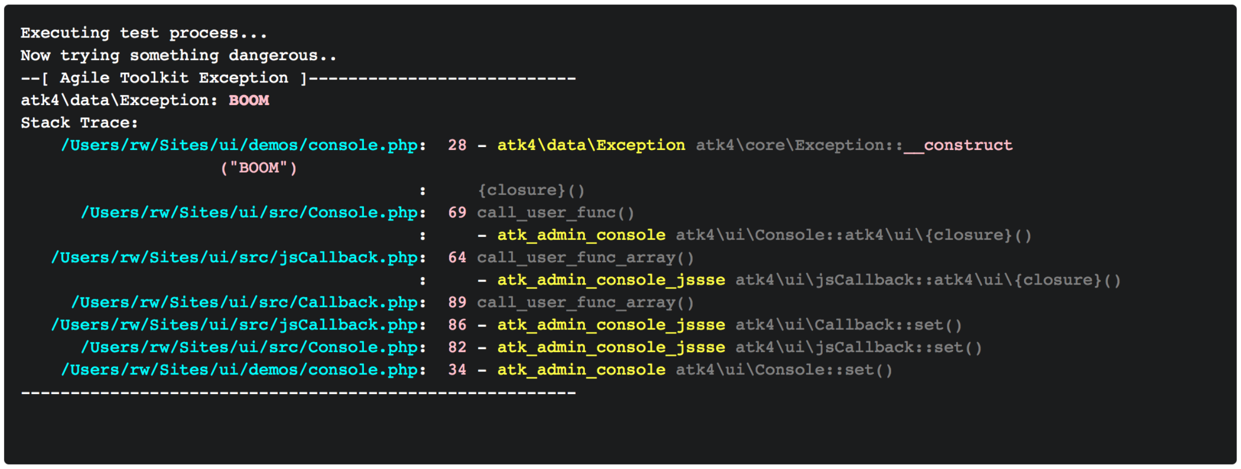
Interactive components rely on *Callbacks*, *VirtualPage* or *Server Sent Event (jsSSE)* to communicate with themselves in the PHP realm. You add them just as you would add any other component, yet they will send additional requests, like loading additional data or executing other code. Here is how interractive components will typically communicate:



1. request by browser is made.
2. *App* asks `Console` to render HTML+JavaScript.
3. JavaScript invokes AJAX request using a *Callback* URL.
4. Callback invokes user-defined PHP code, which will generate some `Console::output()`.
5. Response is encoded and
6. sent back to the browser either as JSON or *Server Sent Event (jsSSE)*.

class atk4\\ui\Console

4.3.1 Console



```
Executing test process...
Now trying something dangerous..
--[ Agile Toolkit Exception ]---------------------------
atk4\data\Exception: BOOM
Stack Trace:
    /Users/rw/Sites/ui/demos/console.php:  28 - atk4\data\Exception atk4\core\Exception::__construct
                ("BOOM")
                                        :     {closure}()
      /Users/rw/Sites/ui/src/Console.php:  69 call_user_func()
                                        :     - atk_admin_console atk4\ui\Console::atk4\ui\{closure}()
   /Users/rw/Sites/ui/src/jsCallback.php:  64 call_user_func_array()
                                        :     - atk_admin_console_jssse atk4\ui\jsCallback::atk4\ui\{closure}()
     /Users/rw/Sites/ui/src/Callback.php:  89 call_user_func_array()
   /Users/rw/Sites/ui/src/jsCallback.php:  86 - atk_admin_console_jssse atk4\ui\Callback::set()
      /Users/rw/Sites/ui/src/Console.php:  82 - atk_admin_console_jssse atk4\ui\jsCallback::set()
    /Users/rw/Sites/ui/demos/console.php:  34 - atk_admin_console atk4\ui\Console::set()
-------------------------------------------------------
```

With console you can output real-time information about the process. We have put a lot of time to capture the output that may be generated by various means and stream it in real-time into HTML component.

Console uses *Server Sent Event (jsSSE)* which works pretty much out-of-the-box with the modern browsers and unlike websockets do not require you to set up additional ports on the server.

Demo: http://ui.agiletoolkit.org/demos/console.php

4.3.1.1 Basic Usage

atk4\\ui\\Console::**set**(*$callback*)

send($callback);

After adding a console to your *Render Tree*, you just need to set a call-back:

```
$console = $app->add('Console');
$console->set(function($console) {

    // This will be executed through SSE request

    $console->output('hello');

    echo 'world'; // also will be redirected to console

    sleep(2);

    $console->send(new \atk4\ui\jsExpression('alert([])', ['The wait is over']);
});
```

Console integrates nicely with DebugTrait (http://agile-core.readthedocs.io/en/develop/debug.html?highlight=debug), and also allows you to execute shell process on the server while redirecting output in real-time.

4.3.1.2 Using With Model

We recommend that you pack up your business logic into your Model methods. When it's time to call your method, you could either do this:

```
$user->generateReport(30);
```

Which would execute your own routine for some report generation, but doing it though a normal request will look like your site is slow and is unable to load page quick. Alternative is to run it through a console:

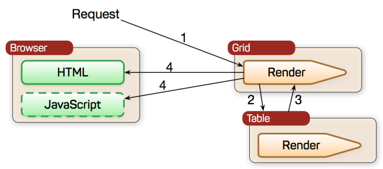
```
$console->setModel($user, 'generateReport', [30]);
```

This will display console to the user and will even output information from inside the model:

```
$this->info('converting report to PDF');
```

4.4 Composite components

Composite elements such as Grid are the bread-and-butter of Agile UI. They will pass on rendering and intractivity to several sub-components. Illustration shows how `Grid` relies on `Table` for rendering the data table, but Grid will also rely on `Menu` and `Paginator` when necessary.



Any component automatically becomes composite if, you use *View::add()*.

4.4.1 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.

4.4.1.1 Using Grid

Here is a simple usage:

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

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

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

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

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

4.4.1.3 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)

4.4.1.4 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;
```

4.4.1.5 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.

$button can be either a string (for a button label) or something like *['icon'=>'book']*.

If $confirm is set to true, then user will see a confirmation when he clicks on the action (yes/no).

Calling this method multiple times will add button into same action column.

See `TableColumnActions::addAction`

Similar to addAction, but when clicking a button, will open a modal dialog and execute $callback to populate a content:

```
$grid->addModalAction('Details', 'Additional Details', function($p, $id) use ($grid) {

    // $id of the record which was clicked
    // $grid->model->load($id);

    $p->add('LoremIpsum');
});
```

Calling this method multiple times will add button into same action column.

See `TableColumnActions::addModal`

4.4.1.6 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()]
));
```

4.4.1.7 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*.

4.4.1.8 Advanced Usage

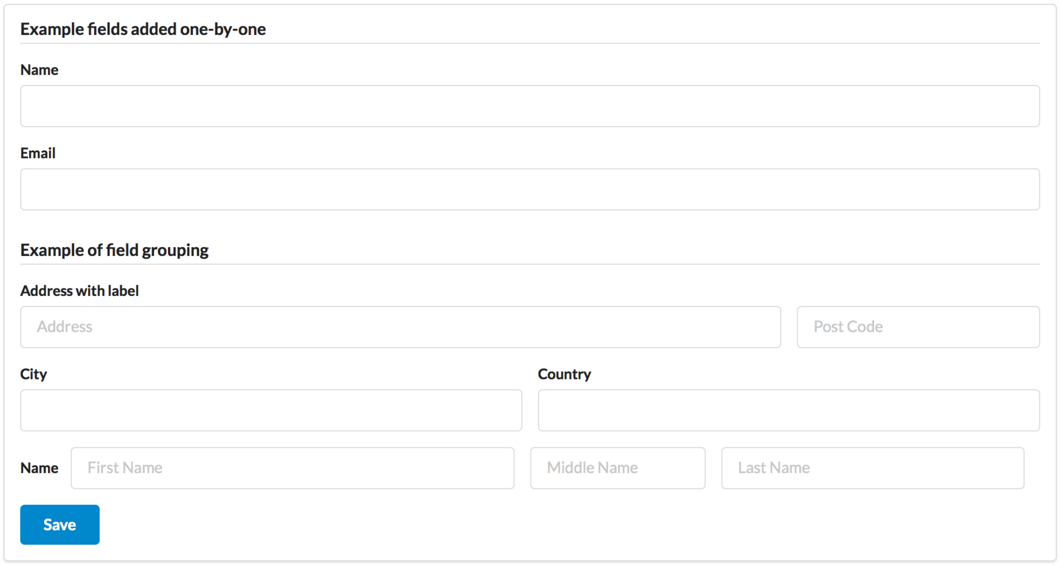
You can use a different component instead of default *Table* by injecting $table property.

4.4.2 Forms

class atk4\ui\Form

One of the most important components of Agile UI is the "Form". Class `Form` implements the following 4 major features:

- Form Rendering using Semantic UI Form (https://semantic-ui.com/collections/form.html):



- Loading and storing data in any database (SQL, NoSQL) supported by Agile Data (http://agile-data.readthedocs.io/en/develop/persistence.html).

- Full Integration with Events and Actions (*JavaScript Mapping*)
- PHP-based Submit Handler using callbacks (*Callbacks*)

Form can be used a web application built entirely in Agile UI or you can extract the component by integrating it into your existing application or framework.

4.4.2.1 Basic Usage

It only takes 2 PHP lines to create a fully working form:

```
$form = $app->add('Form');
$form->addField('email');
```

The form component can be further tweaked by setting a custom call-back handler directly in PHP:

```
$form->onSubmit(function($form) {
    // implement subscribe here

    return "Subscribed ".$form->model['email']." to newsletter.";
});
```

Form is a composite component and it relies on other components to render parts of it. Form uses `Button` that you can tweak to your liking:

```
$form->buttonSave->set('Subscribe');
$form->buttonSave->icon = 'mail';
```

Form also relies on a `atk4\ui\FormLayout` class and displays fields through decorators defined at `atk4\ui\FormField`. See dedicated documentation for:

- `FormLayout::Generic`
- `FormField::Generic`

The rest of this chapter will focus on Form mechanics, such as submission, integration with front-end, integration with Model, error handling etc.

Usage with Model

A most common use of form is if you have a working Model (http://agile-data.readthedocs.io/en/develop/model.html):

```
// Form will automatically add a new user and save into the database
$form = $app->add('Form');
$form->setModel(new User($db));
```

The basic 2-line syntax will extract all the required logic from the Model including:

- Fields defined for this Model will be displayed
- Display of default values in the form
- Depending on field type, a decorator will be selected from FormField/Generic
- Using `FormLayout::Columns` can make form more compact by splitting it into columns
- Field captions, placeholders, hints and other elements defined in Field::ui are respected (http://agile-data.readthedocs.io/en/develop/fields.html#Field::$ui)

- Fields that are not editable by default will not appear on the form (http://agile-data.readthedocs.io/en/develop/fields.html#Field::isEditable)
- Field typecasting will be invoked such as for converting dates
- Reference fields (http://agile-data.readthedocs.io/en/develop/references.html?highlight=hasOne#hasone-reference) displayed as DropDown
- Booleans are displayed as checkboxes but stored as defined by the model field
- Mandatory and Required fields will be visually highlighted (http://agile-data.readthedocs.io/en/develop/fields.html?highlight=required#Field::\protect\T1\textdollarmandatory)
- Validation will be performed and errors will appear on the form (NEED LINK)
- Unless you specify a submission handler, form will save the model `User` into `$db` on successful submission.

All of the above works auto-magically, but you can tweak it even more:

- Provide custom submission handler
- Specify which fields and in which order to display on the form
- Override labels, decorator classes
- Froup fields or use custom layout template
- Mix standard model fields with your own
- Add JS Actions around fields
- Split up form into multiple tabs

If your form is NOT associated with a model, then Form will automatically create a `ProxyModel` and associate it with your Form. As you add fields, they will also be added into ProxyModel.

Extensions

Starting with Agile UI 1.3 Form has a stable API and we expect to introduce some extensions like:

- Capcha decorator
- File Upload field
- Multi-record form
- Multi-tab form

If you develop feature like that, please let me know so that I can include it in the documentation and give you credit.

4.4.2.2 Adding Fields

`atk4\ui\Form::`**`addField`**(*$name*, *$decorator = null*, *$field = null*)

Create a new field on a form:

```
$form = $app->add('Form');
$form->addField('email');
$form->addField('gender', ['DropDown', 'values'=>['Female', 'Male']);
$form->addField('terms', null, ['type'=>'boolean', 'caption'=>'Agree to Terms &␣
↪Conditions']);
```

Create a new field on a form using Model does not require you to describe each field. Form will rely on Model Field Definition and UI meta-values to decide on the best way to handle specific field type:

```
$form = $app->add('Form');
$form->setModel(new User($db), ['email', 'gender', 'terms']);
```

Adding new fields

First argument to addField is the name of the field. You cannot have multiple fields with the same name.

If field exist inside associated model, then model field definition will be used as a base, otherwise you can specify field definition through 3rd argument. I explain that below in more detail.

You can specify first argument `null` in which case decorator will be added without association with field. This will not work with regular fields, but you can add custom decorators such as CAPCHA, which does not really need association with a field.

Field Decorator

To avoid term miss-use, we use "Field" to refer to `\atk4\data\Field`. This class is documented here: http://agile-data.readthedocs.io/en/develop/fields.html

Form uses a small UI components to vizualize HTML input fields associated with the respective Model Field. We call this object "Field Decorator". All field decorators extend from class `FormField::Generic`.

Agile UI comes with at least the following decorators:

- Input (also extends into Line, Password, Hidden)
- DropDown
- CheckBox
- Radio
- Calendar
- Radio
- Money

For some examples see: http://ui.agiletoolkit.org/demos/form3.php

Field Decorator can be passed to `addField` using 'string', *Seed* or 'object':

```
$form->addField('accept_terms', 'CheckBox');
$form->addField('gender', ['DropDown', 'values'=>['Female', 'Male']]);

$calendar = new \atk4\ui\FormField\Calendar();
$calendar->type = 'tyme';
$calendar->options['ampm'] = true;
$form->addField('time', $calendar);
```

For more information on default decorators as well as examples on how to create your own see documentation on `FormField::Generic`.

`atk4\ui\Form::`**`decoratorFactory`**(*atk4dataField $f*, *$defaults* =[ ])

If Decorator is not specified (`null`) then it's class will be determined from the type of the Data Field with `decoratorFactory` method.

Data Field

Data field is the 3rd argument to `Form::addField()`.

There are 3 ways to define Data Field using 'string', 'array' or 'object':

```
$form->addField('accept_terms', 'CheckBox', 'Accept Terms & Conditions');
$form->addField('gender', null, ['enum'=>['Female', 'Male']]);

class MyBoolean extends \atk4\data\Field {
    public $type = 'boolean';
    public $enum = ['N', 'Y'];
}
$form->addField('test2', null, new MyBoolean());
```

String will be converted into `['caption' => $string]` a short way to give field a custom label. Without a custom label, Form will clean up the name (1st argument) by replacing '_' with spaces and uppercasing words (accept_terms becomes "Accept Terms")

Specifying array will use the same syntax as the 2nd argument for `\atk4\data\Model::addField()`. (http://agile-data.readthedocs.io/en/develop/model.html#Model::addField)

If field already exist inside model, then values of $field will be merged into existing field properties. This example make email field mandatory for the form:

```
$form = $app->add('Form');
$form->setModel(new User($db), false);

$form->addField('email', null, ['required'=>true]);
```

addField into Existing Model

If your form is using a model and you add additional field, then it will automatically be marked as "never_persist" (http://agile-data.readthedocs.io/en/develop/fields.html#Field::\protect\T1\textdollarnever_persist).

This is to make sure that custom fields wouldn't go directly into database. Next example displays a registration form for a User:

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

        $this->addField('email');
        $this->addFiled('password');
    }
}

$form = $app->add('Form');
$form->setModel(new User($db));

// add password verification field
$form->addField('password_verify', 'Password', 'Type password again');
$form->addField('accept_terms', null, ['type'=>'boolean']);

// submit event
$form->onSubmit(function($form){
    if ($form->model['password'] != $form->model['password_verify']) {
        return $form->error('password_verify', 'Passwords do not match');
    }

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

    $form->model->save(); // will only store email / password
    return $form->success('Thank you. Check your email now');
});
```

Type vs Decorator Class

Sometimes you may wonder - should you pass decorator class ('CheckBox') or a data field type (['type' => 'boolean']);

I always to recommend use of field type, because it will take care of type-casting for you. Here is an example with date:

```
$form = $app->add('Form');
$form->addField('date1', null, ['type'=>'date']);
$form->addField('date2', ['Calendar', 'type'=>'date']);

$form->onSubmit(function($form) {
    echo 'date1 = '.print_r($form->model['date1'], true).' and date2 = '.print_r(
↪$form->model['date2'], true);
});
```

Field `date1` is defined inside a `ProxyModel` as a date field and will be automatically converted into DateTime object by Persistence typecasting.

Field `date2` has no type and therefore Persistence typecasting will not modify it's value and it's stored inside model as a string.

The above code result in the following output:

```
date1 = DateTime Object ( [date] => 2017-09-03 00:00:00 .. ) and date2 = September 3,␣
↪2017
```

Seeding Decorator from Model

In a large projects, you most likely will not be setting individual fields for each Form, instead you would simply use `addModel()` to populate all defined fields inside a model. Form does have a pretty good guess about Decorator based on field type, but what if you want to use a custom decorator?

This is where `$field->ui` comes in (http://agile-data.readthedocs.io/en/develop/fields.html#Field::\protect\T1\textdollarui).

You can specify `'ui'=>['form' => $decorator_seed]` for your model field:

```
class User extends \atk4\data\Model {
    public $table = 'user';

    function init() {
        parent::init();
        $this->add('email');
        $this->add('password', ['type'=>'password']);

        $this->add('birth_year', ['type'=>'date', 'ui'=>['type'=>'month']);
    }
}
```

The seed for the UI will be combined with the default overriding `FormFieldCalendar::type` to allow month/year entry by the Calendar extension, which will then be saved and stored as a regular date. Obviously you can also specify decorator class:

```
$this->add('birth_year', ['ui'=>['Calendar', 'type'=>'month']);
```

Without the 'type' propoerty, now the calendar selection will be stored as text.

using setModel()

Although there were many examples above for the use of setModel() this method needs a bit more info:

```
.. php:attr:: model
```

`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, so you can then use *addField* to import fields individually.

See also: http://agile-data.readthedocs.io/en/develop/fields.html#Field::isEditable

Loading Values

Although you can set form fields individually using `$form->model['field'] = $value` it's always nicer to load values for the database. Given a `User` model this is how you can create a form to change profile of a currently logged user:

```
$user = new User($db);
$user->getElement('password')->never_persist = true; // ignore password field
$user->load($current_user);

// Display all fields (except password) and values
$form = $app->add('Form');
$form->setModel($user);
```

Submitting this form will automatically store values back to the database. Form uses POST data to submit itself and will re-use the query-string, so you can also safely use any GET arguments for passing record $id. You may also perform model load after record association. This gives the benefit of not loading any other fileds, unless it's marked as System (http://agile-data.readthedocs.io/en/develop/fields.html#Field::\protect\T1\textdollarsystem), see http://agile-data.readthedocs.io/en/develop/model.html?highlight=onlyfields#Model::onlyFields:

```
$form = $app->add('Form');
$form->setModel(new User($db), ['email', 'name']);
$form->model->load($current_user);
```

As before, field `password` will not be loaded from the database, but this time using onlyFields restriction rather then *never_persist*.

Validating

Topic of Validation in web apps is quite extensive. You sould start by reading what Agile Data has to say about validation: http://agile-data.readthedocs.io/en/develop/persistence.html#validation

TL;DR - sometimes validation needed when storing field value inside model (e.g. setting boolean to “blah”) and sometimes validation should be performed only when storing model data into database.

Here are few questions:

- If user specified incorrect value into field, can it be stored inside model and then re-displayed in the field again? If user must enter “date of birth” and he picks date in the future, should we reset field value or simply indicate error?
- If you have a multi-step form with a complex logic, it may need to run validation before record status changes from “draft” to “submitted”.

As far as form is concerned:

- Decorators must be able to parse entered values. For instance DropDown will make sure that value entered is one of the available values (by key)
- Form will rely on Agile Data Typecasting (http://agile-data.readthedocs.io/en/develop/typecasting.html) to load values from POST data and store them in model.
- Form submit handler will rely on `Model::save()` (http://agile-data.readthedocs.io/en/develop/persistence.html#Model::save) not to throw validation exception.
- Form submit handler will also interpret use of *`Form::error`* by displaying errors that do not originate inside Model save logic.

Example use of Model’s validate() method:

```
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:

```
$app->add('Form')
    ->setModel(new Person($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 FormLayout-Generic.

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*.

4.4.3 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 compononents.

4.4.3.1 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 = $app->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
```

4.4.3.2 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 paganiator 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.

4.4.3.3 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.

4.4.3.4 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.

4.4.4 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.

`atk4\ui\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']));
```

4.4.4.1 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 controll 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.

4.4.4.2 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.

CHAPTER 5

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();
```

5.1 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.

5.1.1 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.

5.1.2 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

5.1.3 Extending

Agile UI builds upon the concepts of actions and events in the following ways:

- Action can be any arbitraty 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.

5.1.4 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.

5.2 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
```

5.2.1 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()*.

5.2.2 View to JS integration

We are not building JavaScript code just for the excercise. 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 `Lister` with interractive 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();
// });
```

5.3 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!!
```

5.3.1 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:

```
$sum = new jsExpression('[]+[]', [$h1, $h2]);
$sum = new jsExpression('[0]+[1]', [0=>$h1, 1=>$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.

5.3.2 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.

5.4 Modal

class Modal

Modal::**set**(*callback*)

Modal::**show**()

Modal::**hide**()

This class allows you to open modal dialogs and close them easily. It's based around Semantic UI .modal(), but integrates PHP callback for dynamically producing content of your dialog:

```
$modal = $app->add(['Modal', 'title' => 'Simple title']);
$modal->set(function ($p) use ($modal) {
    $p->add('LoremIpsum');

    $p->add(['Button', 'Hide'])->on('click', $modal->hide());
});

$app->add(['Button', 'Show'])->on('click', $modal->show());
```

Modal will render as a <*div*> block but will be hidden. Alternatively you can use Modal without loadable content:

```
$modal = $app->add(['Modal', 'title' => 'Add a name']);
$modal->add('LoremIpsum');
$modal->add(['Button', 'Hide'])->on('click', $modal->hide());

$app->add(['Button', 'Show'])->on('click', $modal->show());
```

This way it's more convenient for holding static content, such as Terms of Service.

5.5 jsModal

class jsModal

This is alternative implementation to `Modal` and is convenient for situations when you do not know in advance that you migth need to open Dialog box. This class is not a component, but rathen an Action so you mustn't add it into Render Tree:

```
$vp = $app->add('VirtualPage');
$vp->add(['LoremIpsum', 'size' => 2]);

$app->add(['Button', 'Dynamic Modal'])
    ->on('click', new \atk4\ui\jsModal('My Popup Title', $vp->getURL('cut')));
```

If compare this with example for `Modal`, you'll notice that Modal div is always destroyed when you close modal instead of hiding it and then re-created again.

5.6 jsNotify

class jsNotify

jsNotify::**setColor**(*color*)

Implementation for dynamic notifier, which you can use to display operation status:

```
$app->add(['Button', 'Test'])->on(
    'click',
    (new \atk4\ui\jsNotify('Not yet implemented'))
        ->setColor('red')
);
```

A typical use case would be to provide visual feedback of an action after used performs operation inside a Modal window with a Form. When user submits a form, it's Submit handler will close modal, so to leave some feedback to the user jsNotify can display a bar on top of the screen for some time:

```
$modal = $app->add(['Modal', 'Modal Title']);

$modal->set(function ($p) use ($modal) {
    $form = $p->add('Form');
    $form->addField('name', null, ['caption'=>'Add your name']);

    $form->onSubmit(function ($f) use ($modal) {
        if (empty($f->model['name'])) {
            return $f->error('name', 'Please add a name!');
        } else {
            return [
                $modal->hide(),
                new \atk4\ui\jsNotify('Thank you '.$f->model['name'])
            ];
        }
    });
});

$app->add(['Button', 'Open Modal'])->on('click', $modal->show());
```

`jsNotify::`**`setIcon`**`(`*color*`)`

`jsNotify::`**`setTransition`**`(`*openTransition*, *closeTransition*`)`

`jsNotify::`**`setDuration`**`(`*duration*`)`

`jsNotify::`**`setPosition`**`(`*duration*`)`

`jsNotify::`**`setWidth`**`(`*duration*`)`

`jsNotify::`**`setOpacity`**`(`*duration*`)`

You can pass options either as array or by calling methods.

`jsNotify::`**`attachTo`**`(`*view*`)`

Finally you can attach your notification to another view:

```
$jsNotify->attachTo($form);
```

5.7 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 intance completing the form can reload some other view:

```
$m_book = new Book($db);

$f = $app->add('Form');
$t = $app->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.

5.8 Background Tasks

Agile UI has addressed one of the big shortcoming with the PHP language - ability to execute running / background processes. It's best illustrated with example.

Say you need to process a large image, resize, find face, watermark, create thumbnails and store externally. For the average image this could take 5-10 seconds, so you'd like to user updated about the process. There are various ways to do so.

The most basic approach you could probably figure out already:

```
$button = $app->add(['Button', 'Process the image']);
$button->on('click', function() use($button, $image) {

    sleep(1); // $image->resize();
    sleep(1); // $image->findFace();
    sleep(1); // $image->watermark();
    sleep(1); // $image->createThumbnails();

    return $button->js()->text('Success')->addClass('disabled');

});
```

However, it would be nice if you could communicate to the user the progress of your process:

5.8.1 Server Sent Event (jsSSE)

class jsSSE

jsSSE::**send**(*action*)

This class implements ability for your PHP code to send messages to the browser in the middle of the process execution:

```
$button = $app->add(['Button', 'Process the image']);

$sse = $app->add(['jsSSE']);

$button->on('click', $sse->set(function() use($sse, $button, $image) {

    $sse->send($button->js()->text('Processing'));
    sleep(1); // $image->resize();

    $sse->send($button->js()->text('Looking for face'));
    sleep(1); // $image->findFace();

    $sse->send($button->js()->text('Adding watermark'));
    sleep(1); // $image->watermark();

    $sse->send($button->js()->text('Creating thumbnail'));
    sleep(1); // $image->createThumbnails();

    return $button->js()->text('Success')->addClass('disabled');

});
```

The jsSSE component plays a crucial role in some high-level components such as `Console` and `ProgressBar`.

CHAPTER 6

Advanced Topics

6.1 Agile Data

Agile Data is a business logic and data persistance 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 overal clarity.

6.2 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.

6.3 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 importante of testing and all of the Agile UI components come fully tested across multiple browsers. In most cases browser compatibilty is defined by the underlying CSS framework.

With Agile UI we will provide you with a guide how to test your own components.

6.3.1 Unit Tests

You only need to unit-test you 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.

6.3.2 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. instatiate 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.

6.3.3 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 invocie
4. assert through other models e.g. by running client report model.

6.3.4 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.

6.3.5 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

PHP Namespace Index

Index