Agile UI

unknown

Jun 08, 2026

CONTENTS

Contents:

CHAPTER

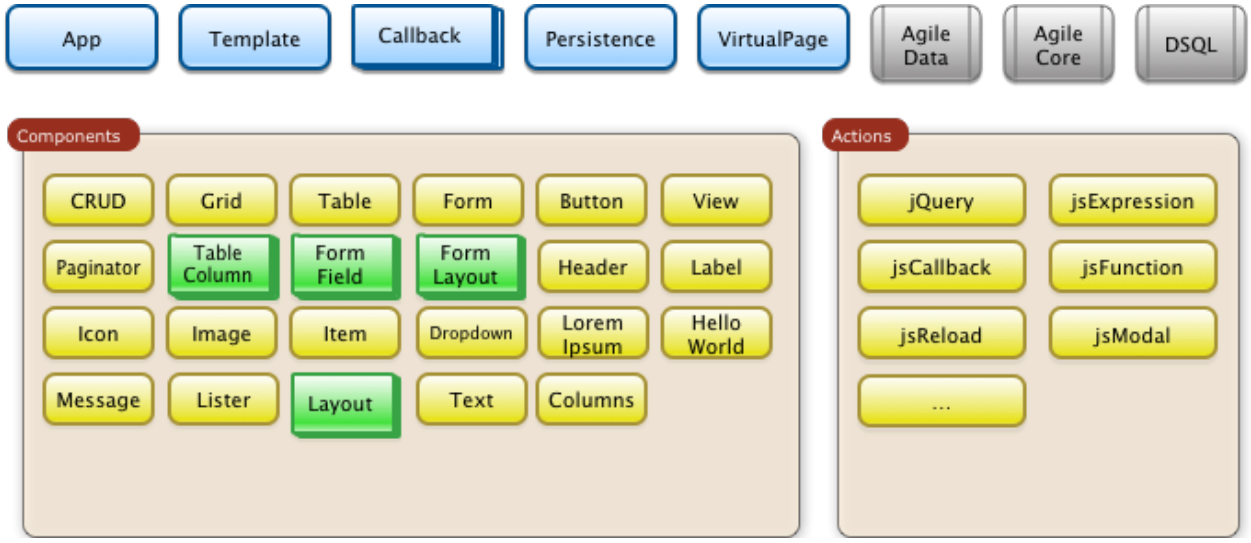
ONE

OVERVIEW OF AGILE UI

Agile UI is a PHP component framework for building User Interfaces entirely in PHP. Although the components of 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 objects.

As a framework it's closely coupled with Agile Data (https://atk4-data.readthedocs.io/) which abstracts database interaction operations. The default UI template set uses Fomantic-UI (https://fomantic-ui.com) for presentation.

At a glance, Agile UI consists of the following:



Agile UI is designed and built for the Agile Toolkit (https://atk4.org/) platform, with the goal of providing a user-friendly experience when creating data-heavy API / UI backends.

1.1 Agile UI Design Goals

Our goal is to offer a free UI framework which can be used to easily develop 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 the .COM boom you purchased 1000 good-looking .COM domains and are now selling them, you will need to track offers from buyers. You could use Excel, but what if your staff needs 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 will 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 an API end-point for authorized access to your offer database, that follows the 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 one hour and then forget about it for the next few years. You are not particularly thrilled about digging through heaps of HTML, CSS, or JS frameworks and need a solution which 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
require_once __DIR__ . '/vendor/autoload.php';

// define your data structure
class Offer extends \Atk4\Data\Model
{
    public $table = 'offer';

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

        $this->addField('domain_name');
        $this->addField('contact_email');
        $this->addField('contact_phone');
        $this->addField('date', ['type' => 'date']);
        $this->addField('offer', ['type' => 'atk4_money']);
        $this->addField('is_accepted', ['type' => 'boolean']);
    }
}

// create Application object and initialize Admin Layout
$app = new \Atk4\Ui\App(['title' => 'Offer tracking system']);
$app->initLayout([\Atk4\Ui\Layout\Admin::class]);

// connect to database and place a fully-interactive Crud
$db = new \Atk4\Data\Persistence\Sql($dsn);
\Atk4\Ui\Crud::addTo($app)
    ->setModel(new Offer($db));
```

Through the course of this example, We are performing several core actions:

- `$app` is an object representing our 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.

We used `Persistence\Sql` class, which takes advantage of a 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 https://atk4-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 our offer records that is then passed into the Crud *Component*.

  For a *Component*, the Model represents information about the structure and offers a 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*, and *Button*.

To sum up Agile UI in more technical terms:

- Fully utilizes abstraction of Web technologies through components.
- Contains concise syntax to define UI layouts in PHP.
- Has built-in security and safety.
- Decouples from data storage/retrieval mechanism.
- 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.

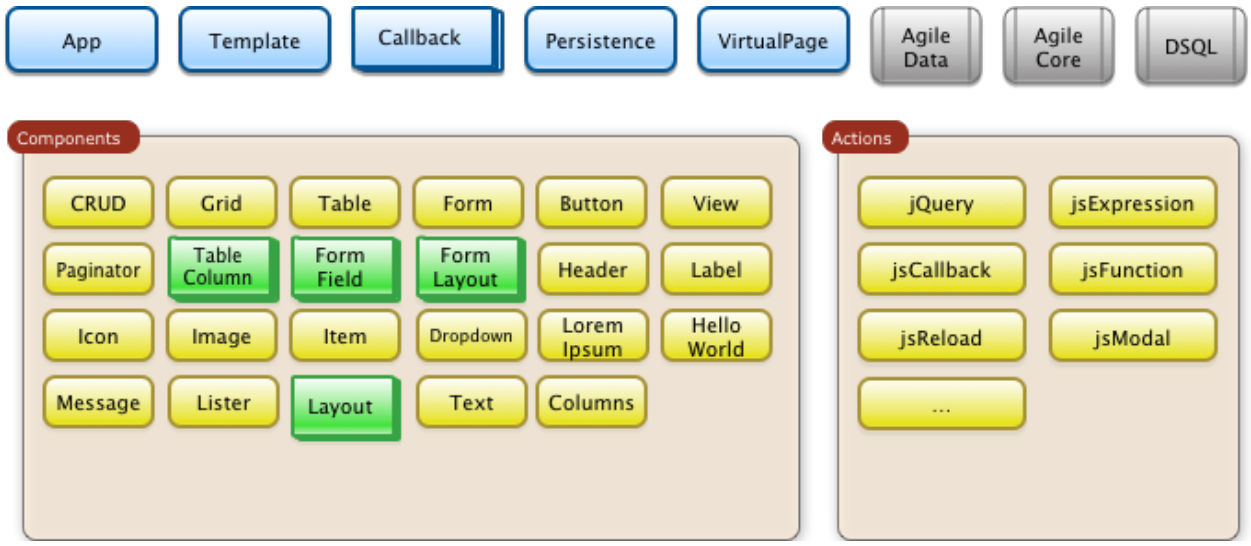
1.2 Component

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 flexibility. The next example adds a "Cancel" button to a form:

```
$button = \Atk4\Ui\Button::addTo($form, [
    '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:

```
\Atk4\Ui\Crud::addTo($app);
```

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

```
$form = new Form();
$form->setApp($app);
$form->invokeInit();
$form->setEntity(new User($db));

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

This would render an individual component and will return HTML:

```
<div class="ui form">
    <form id="atk_form">
       ... fields
       ... buttons
    </form>
</div>
```

For other use-cases please look into *View::renderToHtml()*

1.2.2 Factory

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

```
\Atk4\Ui\Button::addTo($form, ['Cancel', 'icon' => 'pencil'])
    ->link('dashboard.php');
```

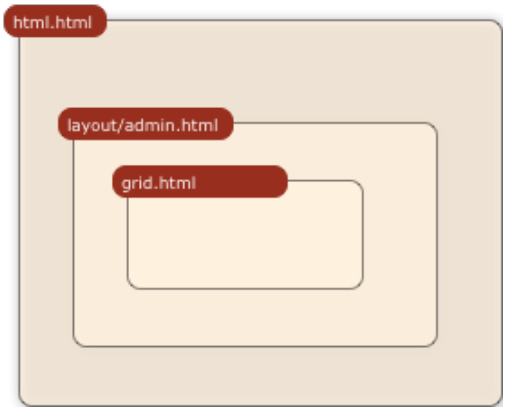
By default, class names specified as the first array elements passed to the add() method are resolved to namespace `Atk4\Ui`; however the application class can fine-tune the search.

Using a factory is optional. For more information see: https://atk4-core.readthedocs.io/en/develop/factory.html

1.2.3 Templates

Components rely on `HtmlTemplate` class for parsing and rendering their HTML. The default template is written for Fomantic-UI framework, which makes sure that elements will look good and be consistent.

1.2.4 Layouts

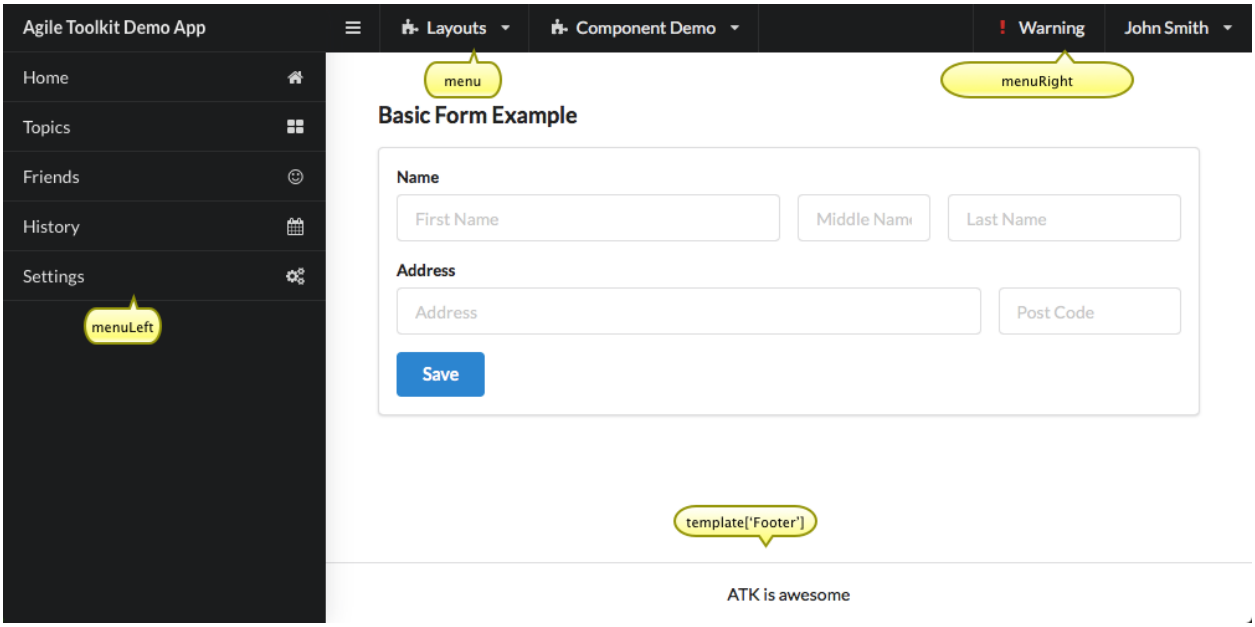


Using App class will utilize a minimum of 2 templates:

- 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 *Layout\Admin* will initialize some components on its own - sidebar menu, top menu.



If you are extending your Admin Layout, be sure to maintain the same property names to allow other components to make use of them. For example, an 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 a 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 component may depend on additional JavaScript libraries, CSS, or other files. At the present time you have to make them available through a CDN and HTTPS. See: `App::requireJs`

1.3.2 Events and Actions

Agile UI allows you to initiate some JavaScript actions from within PHP. The amount of code involvement is quite narrow and is only intended for binding events inside your component without involving developers who use and implement your component.

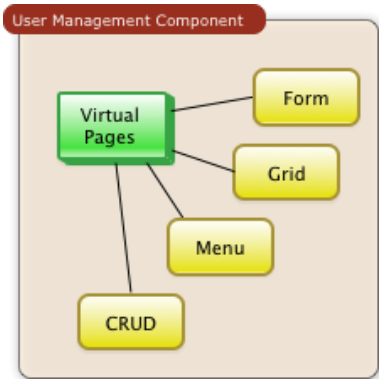
1.3.3 Callbacks

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 any 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 callback.

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 a 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 a component wishes to use a 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. However, Agile Toolkit goes one step further by offering you a directory of published add-ons and installs 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 a very powerful framework for defining data-driven business models and Agile UI offers a very straightforward extension to attach your data to a wide range of standard UI widgets.

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

You can still implement ReactJS applications 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 of the basic apps. You will need to have a basic understanding of "code" and some familiarity with the 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 Table\Column
  - Form and Field
  - Grid and Crud
  - Paginator
- Advanced Topics

If you are not interested in UI and only need the Rest API, we recommend that you look into documentation for Agile Data (https://atk4-data.readthedocs.io/) and the Rest API extension (https://github.com/atk4/api) which is a work in progress.

1.4.2 Application Tutorials

We have written a few working cloud applications ourselves with Agile Toolkit and are offering you to 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, please do not hesitate to reach out.

1.5 Things Agile UI simplifies

Some technologies are "pre-requirements" 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 use 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.

1.6 Hosted Demo showing many functions

There's a demo available of atk4/ui which shows many of the modules and functions available in atk4. You can watch & use the source code of each example to find best practice examples and to see how to use atk4 in certain application cases.

You can find the demo here: https://ui.atk4.org/demos/

1.7 Local Demo/Sandpit

When you download and install atk4 you will find a subdirectory called "demos" in the atk4 repository which also could be locally executed.

1.7.1 Setup the demo

To run the demo:

- Create a directory called “atk4” and create a separate folder for each repo (ui, data, etc.), in this case “ui”
- Fork the original repo into this directory
- Copy the file “db.default.php” from the “atk4/ui/demos” subdirectory
- Rename the copied file to “db.php”
- Open the renamed file and edit your database details accordingly to fit to your database
- Setup an Sqlite file database using a provided script (see below)
- Open the demos from your browser (e.g. https://localhost/…/demos/)

1.7.2 Setup database with example data

The demo also includes a script that let’s you setup a Sqlite file database with an example data. You will find this script in the subdirectory “atk4/ui/demos/_demo-data/”. To run this script, use the following command:

```
php atk4/ui/demos/_demo-data/create-db.php
```

CHAPTER

TWO

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 is 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 which 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"

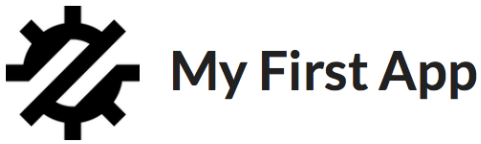
Open a new file `index.php` and enter the following code:

```
<?php
require_once __DIR__ . '/vendor/autoload.php';

$app = new \Atk4\Ui\App(['title' => 'My First App']);
$app->initLayout([\Atk4\Ui\Layout\Centered::class]);

\Atk4\Ui\HelloWorld::addTo($app);
```

Clarifications

- All PHP files start with <?php. I will omit this line in my further examples. There is no need to add a matching ?> at the end.
- Inclusion of `autoload.php` is a standard thing to do when working with PHP / Composer.
- The `App` class represents your web application. This line may change if you integrate Agile UI with another framework.
- Specifies default page layout for your application. Try changing between LayoutCentered and LayoutCentered.
- Creates new component 'HelloWorld' and adds it into Application Layout.

You should see the following output:



Hello World

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

2.4 Using namespaces

By using namespaces you will be able to write less code for classes you use more often by using namespace references and writing clearer code.

By using namespaces you will make out of this:

```
<?php
$app = new \Atk4\Ui\App(['title' => 'My First App']);
```

this:

```
<?php
use \Atk4\Ui\App; // just declared once at the top of your file

$app = new App(['title' => 'My First App']);
```

2.5 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_once __DIR__ . '/vendor/autoload.php';

$app = new \Atk4\Ui\App(['title' => 'ToDo List']);
$app->initLayout([\Atk4\Ui\Layout\Centered::class]);
```

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

```
<?php
session_start();
$s = new \Atk4\Data\Persistence\Array_($_SESSION);
```

If you're establishing a database connection that should be used throughout your whole application and in many classes, you can define it in the $app->db class:

```
<?php
use Atk4\Data\Persistence;
use Atk4\Ui\App;

$db = Persistence::connect(DB_URI, DB_USR, DB_PWD);

$app = new App([
    "title" => "Erp v." . ERP_VER,
    "db" => $db,
    "callExit" => false,
]);
```

2.6 Data Model

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

```
class ToDoItem extends \Atk4\Data\Model
{
    public $table = 'todo_item';

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

        $this->addField('name', ['caption' => 'Task Name', 'required' => true]);

        $this->addField('due', [
            'type' => 'date',
```

(continues on next page)

(continued from previous page)

```
            'caption' => 'Due Date',
            'default' => new \DateTime('+1 week'),
        ]);
    }
}
```

Clarifications

- $table is a default table/collection/key name when persisting model data.
- Second argument to addField() is optional and can contain field meta-data.
- All Meta-data is stored but some has special meaning - 'type' will specify how UI presents the field
- Business Model is always using native PHP types, regardless of where data is stored.

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

2.7 Instantiate App using DiContainerTrait (Dependency Injection)

Class App use `DiContainerTrait` which allow us to inject dependency directly in constructor:

```
use Monolog\Logger;
use Monolog\Handler\StreamHandler;

// create a log channel
$logger = new Logger('name');
$logger->pushHandler(new StreamHandler('path/to/your.log', Logger::WARNING));

use Atk4\Data\Persistence;
use Atk4\Ui\App;
$db = Persistence::connect("mysql://localhost:3306/database_name", "user", "password");

$app = new App([
    "title" => "Your application title",
    "db" => $db,
    "logger" => $logger,
]);
```

2.8 Form and Crud Components

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

```
$col = \Atk4\Ui\Columns::addTo($app, ['divided']);
$jsColReload = new \Atk4\Ui\Js\JsReload($col);

$form = \Atk4\Ui\Form::addTo($col->addColumn());
$form->setEntity(new ToDoItem($s));
$form->onSubmit(function (Form $form) use ($jsColReload) {
```

(continues on next page)

(continued from previous page)

```
    $form->entity->save();

    return $jsColReload;
});

\Atk4\Ui\Table::addTo($col->addColumn())
    ->setModel(new ToDoItem($s));
```

Clarifications

- We wish to position Form and Table side-by-side, so we use `\Atk4\Ui\Columns` component and inject a Fomantic-UI CSS class "divided" that will appear as a vertical separation line.
- $jsColReload is a special object which we call *Actions*. It represents a Browser-event that will cause both columns to be reloaded from the server. To use this action we still have to bind it.
- `Columns` class provides `addColumn()` method to equally divide layout vertically. We call this method twice in our example, so two columns will be visible. Method returns a `View` where we can add a Form component.
- `setModel/setEntity` provides a way to bind Component with Data Model and Data Persistence.
- `Form` relies on a special Callback feature of Agile UI to automatically handle onSubmit callback, pre-load form values into the model, so that you could simply
- Save the record into that session data. Form automatically captures validation errors.
- We use `$jsColReload` which we defined earlier to instruct client browser on what it needs to do when form is successfully saved.
- Very similar syntax to what we used with a form, but using with a Table for listing records.

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.

2.9 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:

```
$grid = \Atk4\Ui\Crud::addTo($col->addColumn(), [
    'paginator' => false,
    'canCreate' => false,
    'canDelete' => false,
]);
$grid->setModel(new ToDoItem($s));

$grid->menu->addItem('Complete Selected',
    new \Atk4\Ui\Js\JsReload($grid->table, [
        'delete' => $grid->addSelection()->jsChecked(),
    ])
);

if ($app->hasRequestQueryParam('delete')) {
```

(continues on next page)

(continued from previous page)

```
    foreach (explode(',', $app->getRequestQueryParam('delete')) as $id) {
        $grid->model->delete($id);
    }
}
```

Clarifications

- We replace 'Table' with a 'Crud'. This is much more advanced component, that wraps 'Table' component by providing support for editing operations and other features like pagination, quick-search, etc.
- Disable create and delete features, since we have other ways to invoke that (form and checkboxes)
- Grid comes with menu, where we can add items.
- You are already familiar with JsReload action. This time we only wish to reload Grid's Table as we wouldn't want to lose any form content.
- Grid's `addSelection` method will add checkbox column. Implemented through `Table\Column\Checkbox` this object has method jsChecked() which will return another Action for collecting selected checkboxes. This demonstrates how Actions can be used as JavaScript expressions augmented by Components.
- Reload events will execute same originating PHP script but will pass additional arguments. In this case, 'delete' get argument is passed.
- We use the IDs to dispose of completed tasks. Since that happens during the Reload event, the App class will carry on with triggering the necessary code to render new HTML for the $grid->table, so it will reflect removal of the items.

2.10 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:

```
ToDoManager::addTo($app)->setModel(new ToDoItem());
```

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

CHAPTER

THREE

CORE CONCEPTS

Agile Toolkit and Agile UI are built upon specific core concepts. Understanding those 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 will always need to have an App class. Even if you do not create this class explicitly, components generally will do it for you. The common pattern is:

```
$app = new \Atk4\Ui\App(['title' => 'My App']);
$app->initLayout([\Atk4\Ui\Layout\Centered::class]);
LoremIpsum::addTo($app);
```

3.1.1 Purpose of App class

class Atk4\Ui**App**

App is a mandatory object that's essential for Agile UI to operate. You should create instance of an App class yourself before other components:

```
$app = new \Atk4\Ui\App(['title' => 'My App']);
$app->initLayout([\Atk4\Ui\Layout\Centered::class]);
LoremIpsum::addTo($app);
```

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 assumptions about the App class, to keep overall code portable.

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

3.1.1.1 Using App for Injecting Dependencies

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

```
$app->db = new \Atk4\Data\Persistence\Sql($dsn);

// later anywhere in the code:

$m = new MyModel($this->getApp()->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 Behavior

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

- using hooks to globally impact object initialization
- override methods to create different behavior, for example url() method may use advanced router logic to create beautiful URLs.
- you may re-define set-up of `Persistence\Ui` 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;

    public function __construct(bool $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']);
        $this->db->setApp($this);

        // my App class provides access to a currently logged user and currently␣
↪selected system
        session_start();

        // App class may be used for pages that do not require authentication
        if (!$auth) {
```

(continues on next page)

(continued from previous page)

```
            $this->initLayout([\Atk4\Ui\Layout\Centered::class]);

            return;
        }

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

        // make sure user is valid
        if ($this->user === null) {
            $this->initLayout([\Atk4\Ui\Layout\Centered::class]);
            Message::addTo($this, ['Login Required', 'type' => 'error']);
            Button::addTo($this, ['Login', 'class.primary' => true])->link('index.php');
            exit;
        }

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

        $this->initLayout([\Atk4\Ui\Layout\Admin::class]);

        // 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';
$app = new Warehouse();
Crud::addTo($app)
    ->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:

```
HelloWorld::addTo($app);
```

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 behavior 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

Atk4\Ui\App::**run**()

property Atk4\Ui\App::$**runCalled**

property Atk4\Ui\App::$**isRendering**

property Atk4\Ui\App::$**alwaysRun**

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:

```
HelloWorld::addTo($app);

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

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

3.1.1.6 Exception handling

Atk4\Ui\App::**caugthException**()

property Atk4\Ui\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 Layout\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.

Atk4\Ui\App::**requireJs**()

Method to include additional JavaScript file in page:

```
$app->requireJs('https://example.com/file.min.js');
```

Atk4\Ui\App::**requireCss**(*$url*)

Method to include additional CSS style sheet in page:

```
$app->requireCss('https://example.com/file.min.css');
```

Atk4\Ui\App::**initIncludes**()

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

3.1.1.8 Loading Templates for Views

`Atk4\Ui\App::`**`loadTemplate`**`(`*`$name`*`)`

Views use *`View::$defaultTemplate`* to specify which template they are using. By default those are loaded from `vendor/atk4/ui/template` however by overriding this method, you can specify extended logic.

You may override this method if you are using a different CSS framework.

3.1.2 Utilities by App

App provides various utilities that are used by other components.

`Atk4\Ui\App::`**`getTag`**`()`

`Atk4\Ui\App::`**`encodeHtml`**`()`

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

3.1.2.1 Sticky GET Arguments

`Atk4\Ui\App::`**`stickyGet`**`()`

`Atk4\Ui\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 the `Crud` component is interactive, it may want to generate requests to itself, but it must also include `order_id` otherwise the scope will be incomplete. Agile UI solves that with StickyGet arguments:

```
$orderId = $app->stickyGet('order_id');
$crud->setModel($order->load($orderId)->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()`*.

See also `View::stickyGet`

3.1.2.2 Redirects

`Atk4\Ui\App::`**`redirect`**`(`*`page`*`)`

`Atk4\Ui\App::`**`jsRedirect`**`(`*`page`*`)`

App implements two handy methods for handling redirects between pages. The main purpose for those is to provide a simple way to redirect for users who are not familiar with JavaScript and HTTP headers so well. Example:

```
if (!$app->hasRequestQueryParam('age')) {
    $app->redirect(['age' => 18]);
}

Button::addTo($app, ['Increase age'])
    ->on('click', $app->jsRedirect(['age' => $app->getRequestQueryParam('age') + 1]));
```

No much magic in these methods.

3.1.2.3 Database Connection

property Atk4\Ui\App::$**db**

If your App needs a DB connection, set this property to an instance of Persistence.

Example:

```
$app->db = \Atk4\Data\Persistence::connect('mysql://user:pass@localhost/atk');
```

See Persistence::connect

3.1.2.4 Execution Termination

Atk4\Ui\App::**terminate**(*output*)

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

3.1.2.5 Execution state

property Atk4\Ui\App::$**isRendering**

Will be true if the application is currently rendering recursively through the render tree.

3.1.2.6 Links

Atk4\Ui\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 behavior 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`.

Atk4\Ui\App::**jsUrl**(*callback_page*)

Use jsUrl for creating callback, which return non-HTML output. This may be routed differently by a host framework (https://github.com/atk4/ui/issues/369).

3.1.2.7 Includes

`Atk4\Ui\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.

3.1.2.8 Hooks

Application implements HookTrait (https://atk4-core.readthedocs.io/en/develop/hook.html) and the following hooks are available:

- beforeRender
- beforeOutput
- beforeExit

Hook beforeExit is called just when application is about to be terminated. If you are using *App::$alwaysRun* = true, then this hook is guaranteed to execute always after output was sent. ATK will avoid calling this hook multiple times.

Note: beforeOutput and beforeRender are not executed if $app->terminate() is called, even if parameter is passed.

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

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

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 (https://atk4-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(['title' => 'My App']);
$app->add($grid);

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

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([\Atk4\Ui\Layout\Centered::class]);
```

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 - Layout\Centered, Layout\Admin etc. This will not only change the overall page outline, but will also introduce some additional views.

Each layout, depending on it's content, may come with several views that you can populate.

3.1.3.3 Admin Layout

class Atk4\Ui**Layout\Admin**

Agile Toolkit comes with a ready to use admin layout for your application. The layout is built with top, left and right menu objects.

property Atk4\Ui\Layout\Admin::$**menuLeft**

Populating the left menu object is simply a matter of adding the right menu items to the layout menu:

```
$app->initLayout([\Atk4\Ui\Layout\Admin::class]);
$layout = $app->layout;

// add item into menu
$layout->menuLeft->addItem(['Welcome Page', 'icon' => 'gift'], ['index']);
$layout->menuLeft->addItem(['Layouts', 'icon' => 'object group'], ['layouts']);

$EditGroup = $layout->menuLeft->addGroup(['Edit', 'icon' => 'edit']);
$EditGroup->addItem('Basics', ['edit/basic']);
```

property Atk4\Ui\Layout\Admin::$**menu**

This is the top menu of the admin layout. You can add other item to the top menu using:

```
Button::addTo($layout->menu->addItem(), ['View Source', 'class.teal' => true, 'icon' =>
↪'github'])
    ->setAttr('target', '_blank')
    ->on('click', new \Atk4\Ui\Js\JsExpression('window.location = []', [$url . $f]));
```

property Atk4\Ui\Layout\Admin::$**menuRight**

The top right dropdown menu.

property Atk4\Ui\Layout\Admin::$**isMenuLeftVisible**

Whether or not the left menu is open on page load or not. Default is true.

3.1.3.4 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.

3.1.3.5 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 Fomantic-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 PHP's dynamic nature, therefore two syntax patterns are supported everywhere:

```
Button::addTo($app, ['Hello']);

and

Button::addTo($app, ['Hello']);
```

Method add() supports arguments in 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

Agile UI relies on wide variety of objects. For example *Button* relies on *Icon* object for its rendering. As a developer can create Icon object first, then pass it to the button:

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

or you can divert icon creation until later by using Array / String for *Button::$icon* property:

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

When you don't provide an object - string/array value is called "Seed" and will be used to locate and load class dynamically just when it's needed.

Seed has many advantages:

- more readable and shorter syntax
- easier concept for new developers and non-developers
- can be namespace-specific
- can improve performance - not all seeds are initialized
- recursive syntax with property and constructor argument injection
- allow App logic to further enhance mechanics

3.2.1.1 Growing Seed

To grow a seed you need a factory. Factory is a trait implemented in atk4/core and used by all ATK UI classes:

```
$object = Factory::factory($seed);
```

In most cases you don't need to call factory yourself, methods which accept object/seed combinations will do it for you:

```
Button::addTo($app);
// app will create instance of class \Atk4\Ui\Button
```

3.2.1.2 Seed, Object and Render Tree

When calling *View::add()* not only your seed becomes an object, but it is also added to the render tree.

3.2.2 Seed Components

For more information about seeds, merging seeds, factories and namespaces, see https://atk4-core.readthedocs.io/.

The most important points of a seed such as this one:

```
$seed = [Button::class, 'hello', 'class.big red' => true, 'icon' => ['book', 'red']];
```

are:

- Element with index 0 is name of the class mapped into namespace \Atk4\Ui (by default).

- Elements with numeric indexes 'hello' and 'big red' are passed to constructor of Button
- Elements with named arguments are assigned to properties after invocation of constructor

3.2.2.1 Alternative ways to use Seed

Some constructors may accept array as the first argument. It is also treated as a seed but without class (because class is already set):

```
$button = new Button(['hello', 'class.big red' => true, 'icon' => ['book', 'class.red' =>
↪ true]]);
```

It is alternatively possible to pass object as index 0 of the seed. In this case constructor is already invoked, so passing numeric values is not possible, but you still can pass some property values:

```
$seed = [new Button('hello', 'class.big red' => true), 'icon' => ['book', 'class.red' =>␣
↪true]];
```

3.2.2.2 Additional cases

An individual object may add more ways to deal with seed. For example, when adding columns to your Table you can specify seed for the decorator: *Table\Column*:

```
$table->addColumn('salary', [\Atk4\Ui\Table\Column\Money::class]);

// or

$table->addColumn('salary', [\Atk4\Ui\Table\Column\Money::class]);

// or

$table->addColumn('salary', new \Atk4\Ui\Table\Column\Money());

// or

$table->addColumn('salary', [new \Atk4\Ui\Table\Column\Money()]);
```

Note that addColumn uses default namespace of `\Atk4\Ui\Table\Column` when seeding objects. Some other methods that use seeds are:

- *Table::addColumn()*
- *Form::addControl()*

3.3 Render Tree

Agile Toolkit 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:

```
$msg = new \Atk4\Ui\Message('Hey There');
Button::addTo($msg, ['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();

\Atk4\Ui\Button::addTo($view, ['test']);

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

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

Here is a breakdown of how the above code works:

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

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

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

Most of the UI classes will allow you to operate even if they are not initialized. For instance calling `setModel()/setEntity()` will simply set a $model/$entity 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 the init() method of a view is essential before any meaningful work can be done with it. This is important, because the following actions are performed:

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

Many UI components rely on the above to function properly. For example, Grid will look for certain regions in its 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->initLayout([\Atk4\Ui\Layout\Centered::class]);

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

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

```
$b = \Atk4\Ui\Button::addTo($app, ['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();
$b2 = \Atk4\Ui\Button::addTo($v, ['Test2']);

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

At this point, if you execute $v->renderToHtml() it will create it's own App and will create its own render tree. On the 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->renderToHtml() will create its 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 $subView.
2. Set $subView->name = false;
3. Calls $view->add($subView);
4. executes $subView->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 $subView will be automatically embedded into the column and having it appear anywhere else on the page would be troublesome.

It's futile to try and extract JS chains from the $subView because JS wouldn't work anyway, 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:

```
$b = \Atk4\Ui\Button::addTo($app);
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. Ultimately, you can create a View that uses it's name to store some information:

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

        if ($this->getApp()->getRequestQueryParam($this->name)) {
            \Atk4\Ui\Label::addTo($this, [
                'Secret info is',
                'class.big red' => true,
                'detail' => $this->getApp()->getRequestQueryParam($this->name),
            ]);
        }

        \Atk4\Ui\Button::addTo($this, ['Send info to ourselves'])
            ->link([$this->name => 'secret_info']);
    }
}
```

This quality of Agile UI objects is further explored through *Callback* and *VirtualPage*

3.4 Sticky GET

Agile UI implements advanced approach allowing any View object that you add into render tree to declare "sticky GET arguments". Here is example:

```
if ($app->hasRequestQueryParam('message')) {
    Message::addTo($app)->set($app->getRequestQueryParam('message'));
}

Button::addTo($app, ['Trigger message'])->link(['message' => 'Hello World']);
```

The code is simple - if you click the button, page will appear with the message just above, however there is a potential problem here. What if "Message" wanted to perform a *Callbacks*? What if we use `Console` instead, which must display an interactive data stream?

In Agile UI you can request that some $_GET arguments are preserved and included into callback urls:

```
if ($this->getApp()->stickyGet('message')) {
    Message::addTo($app)->set($app->getRequestQueryParam('message'));
}

Button::addTo($app, ['Trigger message'])->link(['message' => 'Hello World']);
```

There are two types of "sticky" parameters, application-wide and view-specific.

3.4.1 Introduction

Ability to automatically generate callback URLs is one of the unique features in Agile UI. With most UI widgets they would rely on a specific URL to be available or would require you to define them.

With Agile UI the backend URLs are created dynamically by using unique names and callbacks.

There is one problem, however. What if View (and the callbacks too) are created conditionally?

The next code creates Loader area which will display a console. Result is - nested callback:

```
Loader::addTo($app)->set(function (Loader $p) {
    Console::addTo($p)->set(function (Console $console) {
        $console->output('success!');
    });
});
```

What if you need to pass a variable `client_id` to display on console output? Technically you would need to tweak the callback URL of "Loader" and also callback URL of "Console".

Sticky GET is a better approach. It works like this:

```
$app->stickyGet('client_id');

Loader::addTo($app)->set(function (Loader $p) {
    Console::addTo($p)->set(function (Console $console) {
        $console->output('client_id = !' . $console->getApp()->getRequestQueryParam(
↪'client_id'));
    });
});
```

Whenever Loader, Console or any other component generatens a URL, it will now include value of `$_GET['client_id']` and it will transparently arrive inside your code even if it takes multiple requests to get there.

3.4.1.1 Global vs Local Sticky GET

In earlier example, we have called `$app->stickyGet` which creates a global stickyGet. After executing, all the invocations to App::url() or View::url() will contain "client_id".

In some cases, Sticky GET only make sense within a certain branch of a render tree. For instance, when Loader wishes to load content dynamically, it must pass extra _GET parameter to trigger a *Callback*. Next, when Console needs to establish live SSE stream, it should include the SAME get argument to trigger a callback for the Loader, otherwise Console wouldn't be initialized at all.

Loader sets a local stickyGet on the $p before it's passed inside your function:

```
$p->stickyGet('trigger_name');
```

This way - all the views added into this $p will carry an extra get argument:

```
$p->url(); // includes "trigger_name=callback"
```

If you call `$app->url()` it will contain `client_id` but won't contain the callbacks triggers.

3.4.1.2 View Reachability

Agile UI views have a method View::url() which will return URL that is guaranteed to trigger their "init" method. This is regardless of the placement of your View and also it honors all the arguments that are defined as sticky globally.

Consider this code:

```
$b1 = \Atk4\Ui\Button::addTo($app);
$b1->set($b1->url());

Loader::addTo($app)->set(function (Loader $p) {
    $b2 = \Atk4\Ui\Button::addTo($p);
    $b2->set($b2->url());
});

$b3 = \Atk4\Ui\Button::addTo($app);
$b3->set($b3->url());
```

This will display 3 buttons and each button will contain a URL which needs to be opened in order for corresponding button to be initialized. Because middle button is inside a callback the URL for that will be different.

3.4.1.3 Dropping sticky argument

Sometimes you want to drop a sticky argument. If your sticky was set locally, you can drop it by calling either a parent's url or $app->url(), however for global sticky Get you can use either `url(['client_id' => false])` or `stickyForget('client_id')`.

3.5 Type Presentation

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

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

3.5.1 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 the distinction between data Presentation and Decoration.

- Data Presentation: displaying value of the data in a different format, e.g. 123,123.00 vs 123.123,00
- Data Decoration: adding currency symbol or calendar icon.

Agile UI believes that 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 control icon to indicate currency.

Presentation in Agile Toolkit is handled by `Persistence\Ui`.

Decoration is performed by helper classes, such as `Form\Control\Calendar` or *Table\Column\Money*. The decorator is in control of the final output, so it can decide if it uses the value from presentation or do some decoration on its own.

3.5.2 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 in `$app->uiPersistence`.

   You need to define how to output your data as well as read it.

2. Try your new type with a standard Form control.

   The value you output should read and stored back correctly. This ensures that standard 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 `Form\Control\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.

   This happens in *Form::controlFactory* and `Table::decoratorFactory`.

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

3.5.3 Manually Specifying Decorators

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

```
$table->addColumn('field_name', new \Atk4\Ui\Table\Column\Password());

// or

$form->addControl('field_name', new \Atk4\Ui\Form\Control\Password());
```

Selecting the decorator is done in the following order:

- specified in second argument to UI `addColumn()` or `addControl()` (as shown above)
- specified using `ui` property of `Atk4\Data\Field`:

  ```
  $field->ui['form'] = new \Atk4\Ui\Form\Control\Password();
  ```

- fallback to *Form::controlFactory*

Note: When talking about "fields": you need to know what kind of field you are talking about (Data or UI). Both **models** (Data) as well as some **views** (UI: form) use fields. They are not the same. Notably, Model field `ui` property contains flags like editable, visible and hidden, which do have some impact on rendering, whereas UI field `ui` property (not used here) designates the Fomantic-UI element to use.

3.5.4 Examples

Let's explore various use cases and how to properly deal with scenarios

3.5.4.1 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:

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

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

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

3.5.4.2 Hide account_number in 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:

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

$table->setModel($model);
$model->addDecorator('account_number', new \Atk4\Ui\Table\Column\Password());
```

3.5.4.3 Create a decorator for hiding credit card number

If you happen to store card numbers and you only want to display the last digits in tables, yet make it available when editing, you could create your own *Table\Column* decorator:

```
class Masker extends \Atk4\Ui\Table\Column
{
    public function getDataCellTemplate(\Atk4\Data\Field $field = null): string
    {
        return '**** **** **** {$mask}';
    }

    public function getHtmlTags(\Atk4\Data\Model $row, ?\Atk4\Data\Field $field): array
    {
        return [
            'mask' => substr($field->get($row), -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::$useHtmlTags`.

3.5.4.4 Display credit card number with spaces

If we always have to display card numbers with spaces, e.g. "1234 1234 1234 1234" but have the database store them without spaces, then this is a data formatting task best done by extending `Persistence\Ui`:

```
class MyPersistence extends Persistence\Ui
{
    protected function _typecastSaveField(\Atk4\Data\Field $field, $value)
    {
        switch ($field->type) {
            case 'card':
                $parts = str_split($value, 4);

                return implode(' ', $parts);
        }

        return parent::_typecastSaveField($field, $value);
    }
```

(continues on next page)

(continued from previous page)

```
    public function _typecastLoadField(\Atk4\Data\Field $field, $value)
    {
        switch ($field->type) {
            case 'card':
                return str_replace(' ', '', $value);
        }

        return parent::_typecastLoadField($field, $value);
    }
}

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

        parent::__construct($defaults);
    }
}
```

Now your 'card' type will work system-wide.

3.6 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 behavior see:

3.6.1 Introduction

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

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

3.6.2 Example Template

Assuming that you have the following template:

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

3.6.2.1 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 renderToHtml() it any-time to get string "Hello, world". You can also change tag value:

```
$t->set('mytag', 'Agile UI');

echo $t->renderToHtml(); // "Hello, Agile UI"
```

Tags may also be self-closing:

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

is identical to:

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

3.6.2.2 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 delegate rendering to another View (more about this in section for Views).

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

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

$mainTemplate->del('Content');

$content->set('user', 'Joe')->set('amount', 100);
$mainTemplate->dangerouslyAppendHtml('Content', $content->renderToHtml());

$content->set('user', 'Billy')->set('amount', 50);
$mainTemplate->dangerouslyAppendHtml('Content', $content->renderToHtml());
```

3.6.2.3 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->template->set('name', 'Mr. Boss');

$lister = new Lister($v, 'Content');
$lister->setModel($userlist);

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

The code above will work like this:

1. View will load and parse template.
2. Using $v->template->set('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.

3.6.3 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 Atk4\Ui**Template**

3.6.3.1 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:

Atk4\Ui\Template::**__construct**(*$templateString*)

Will parse template specified as an argument.

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

Atk4\Ui\Template::**loadFromFile**(*$filename*)

Read file and load contents as a template.

Atk4\Ui\Template::**tryLoadFromFile**(*$filename*)

Try loading the template. Returns false if template couldn't be loaded. This can be used if you attempt to load template from various locations.

Atk4\Ui\Template::**loadFromString**(*$string*, *bool $allowParseCache = false*)

Same as using constructor.

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

3.6.3.2 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 PascalCase, 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.

3.6.3.3 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 HtmlTemplate::loadTemplate or HtmlTemplate::loadFromString.
2. Set tag and region values with HtmlTemplate::set.
3. Render template with HtmlTemplate::renderToHtml.

3.6.3.4 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\Control` 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()
Filters (to-upper, escape)	all tags are escaped automatically, but other filters are not supported (yet)

3.6.4 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.

3.6.4.1 Loading template

`Atk4\Ui\Template::`**loadFromString**(*string*)

Initialize current template from the supplied string

`Atk4\Ui\Template::`**loadFromFile**(*filename*)

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

`Atk4\Ui\Template::`**__clone**()

Will create duplicate of this template object.

property `Atk4\Ui\Template::$`**template**

Array structure containing a parsed variant of your template.

property `Atk4\Ui\Template::$`**tags**

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

property `Atk4\Ui\Template::$`**template_source**

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

property `Atk4\Ui\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:

```
$template = HtmlTemplate::addTo($this);
$template->loadFromString('Hello, {name}world{/}');
```

To load template from file:

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

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

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

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

3.6.4.2 Changing template contents

`Atk4\Ui\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.

`Atk4\Ui\Template::`**`dangerouslySetHtml`**(*tag*, *value*)

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

`Atk4\Ui\Template::`**`append`**(*tag*, *value*)

Escape and add value to existing tag.

`Atk4\Ui\Template::`**`tryAppend`**(*tag*, *value*)

Attempts to append value to existing but will do nothing if tag does not exist.

`Atk4\Ui\Template::`**`dangerouslyAppendHtml`**(*tag*, *value*)

Similar to append, but will not escape.

`Atk4\Ui\Template::`**`tryDangerouslyAppendHtml`**(*tag*, *value*)

Attempts to append non-escaped value, but will do nothing if tag does not exist.

Example:

```
$template = HtmlTemplate::addTo($this);
$template->loadFromString('Hello, {name}world{/}');

$template->set('name', 'John');
$template->dangerouslyAppendHtml('name', ' <i class="icon-heart"></i>');

echo $template->renderToHtml();
```

Using ArrayAccess with Templates

You may use template object as array for simplified syntax:

```
$template->set('name', 'John');

if ($template->hasTag('has_title')) {
    $template->del('has_title');
}
```

3.6.4.3 Rendering template

`Atk4\Ui\Template::renderToHtml()`
: 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:

```
$html = $template->renderToHtml();

\Atk4\Ui\Text::addTo($this)->dangerouslyAddHtml($html);
// will output "Hello, World"
```

3.6.4.4 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:

```
$template = HtmlTemplate::addTo($this);
$template->loadFromFile('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
$sender->set($senderData);
$recipient->set($recipientData);

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

(continues on next page)

(continued from previous page)

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

Same thing using Agile Toolkit Views:

```
$envelope = \Atk4\Ui\View::addTo($this, [], [null], null, ['envelope']);

$sender = \Atk4\Ui\View::addTo($envelope, [], [null], 'Sender', 'Sender');
$recipient = \Atk4\Ui\View::addTo($envelope, [], [null], 'Recipient', 'Recipient');

$sender->template->set($senderData);
$recipient->template->set($recipientData);
```

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

In this example I've 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 `renderToHtml()` method.

3.6.4.5 Other operations with tags

`Atk4\Ui\Template::`**`del`**`(`*tag*`)`

Empties contents of tag within a template.

`Atk4\Ui\Template::`**`hasTag`**`(`*tag*`)`

Returns `true` if tag exists in a template.

`Atk4\Ui\Template::`**`trySet`**`(`*name*, *value*`)`

Attempts to set a tag, if it exists within template

`Atk4\Ui\Template::`**`tryDel`**`(`*name*`)`

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

3.6.4.6 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.

3.6.4.7 Conditional tags

Agile Toolkit template engine allows you to use so called conditional tags which will automatically remove template regions if tag value is empty. Conditional tags notation is trailing question mark symbol.

Consider this example:

```
My {email?}e-mail {$email}{/email?} {phone?}phone {$phone}{/?}.
```

This will only show text "e-mail" and email address if email tag value is set to not empty value. Same for "phone" tag. So if you execute `set('email', null)` and `set('phone', 123)` then this template will automatically render as:

```
My  phone 123.
```

Note that zero value is treated as not empty value!

3.6.5 Views and Templates

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

3.6.5.1 Default template for a view

`Atk4\Ui\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 `loadFromFile()`. 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:

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

3.6.5.2 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:

```
MyObject::addTo($this, ['greeting']);
```

3.6.5.3 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:

```
$grid = \Atk4\Ui\Grid::addTo($this, [], [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.

3.6.5.4 How views render themselves

Agile Toolkit perform object initialization first. When all the objects are initialized global rendering takes place. Each object's `renderToHtml()` 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.

```
$envelope = \Atk4\Ui\View::addTo($this, [], [null], null, array('envelope'));

// 3rd argument is output region, 4th is template location
$sender = \Atk4\Ui\View::addTo($envelope, [], [null], 'Sender', 'Sender');
$receiver = \Atk4\Ui\View::addTo($envelope, [], [null], 'Receiver', 'Receiver');

$sender->template->trySet($senderData);
$receiver->template->trySet($receiverData);
```

3.6.6 Best Practices with Views

3.6.6.1 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 HtmlTemplate 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)

3.6.6.2 Organize templates into directories

Typically templates directory will have sub-directories: "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.

3.6.6.3 Naming of tags

Tags use two type of naming - PascalCase 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 PascalCase. 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".

3.6.7 Globally Recognized Tags

Agile Toolkit View will automatically substitute several tags with the values. The tag {$attributes} is automatically replaced with a attributes incl. `id` (unique name of a View), `class` and `style`.

There are more templates which are being substituted:

- {page}logout{/} - will be replaced with relative URL to the page
- {public}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

Application (API) has a function `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.

3.7 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 = $user->load(1);

$view = View::addTo($app, ['template' => 'Hello, {$name}, your balance is {$balance}']);
$view->setEntity($user);
```

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

3.7.1 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.

3.7.2 Static Data Arrays

Agile Data contains Persistence\Array_ (https://atk4-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.

3.7.3 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:

```
// 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.

3.8 Callbacks

By relying on the ability of generating *Unique Name*, it's possible to create several classes for implementing PHP callbacks. 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';
});
```

3.8.1 Callback Introduction

Agile UI pursues a goal of creating a full-featured, interactive, 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 behavior 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 (Jquery $j) {
    return $j->text(rand(1, 100));
});
```

This creates callback 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.

3.8.2 The Callback class

class Atk4\Ui**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:

Atk4\Ui\Callback::**getUrl**(*$val*)

Atk4\Ui\Callback::**set**()

The following example code generates unique URL:

```
$label = \Atk4\Ui\Label::addTo($app, ['Callback URL:']);
$cb = \Atk4\Ui\Callback::addTo($label);
$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 `Callback::set()`:

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

3.8.3 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 behavior (of rendering UI) and will output specified string instead, stopping further execution of your application.

3.8.4 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:

```
$label = \Atk4\Ui\Label::addTo($app, ['Callback URL:']);
$cb = \Atk4\Ui\Callback::addTo($label);
$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 *Callback::$triggered*.

property Atk4\Ui\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 callback link. The next example will change color of the label regardless of the callback function:

```
$label = \Atk4\Ui\Label::addTo($app, ['Callback URL:']);
$cb = \Atk4\Ui\Callback::addTo($label);
$label->detail = $cb->getUrl();
$label->link($cb->getUrl());

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

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

property Atk4\Ui\Callback::$**postTrigger**

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.

property Atk4\Ui\Callback::$**urlTrigger**

Specifies which GET parameter to use for triggering. Normally it's same as `$callback->name`, but you can set it to anything you want. As long as you keep it unique on a current page, you should be OK.

3.8.5 CallbackLater

class Atk4\Ui**CallbackLater**

This class is very similar to Callback, but it will not execute immediately. 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:

```
$label = \Atk4\Ui\Label::addTo($app, ['Callback URL:']);
$cb = \Atk4\Ui\CallbackLater::addTo($label);

$cb->set(function () use ($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 *Js\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:

```
$view = \Atk4\Ui\View::addTo($app, ['ui' => 'tertiary green inverted segment']);
$button = \Atk4\Ui\Button::addTo($app, ['Reload Lorem']);

$button->on('click', new \Atk4\Ui\Js\JsReload($view));

\Atk4\Ui\LoremIpsum::addTo($view);
```

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 existence is only possible thanks to CallbackLater implementation.

3.8.6 JsCallback

class Atk4\Ui**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:

```
$label = \Atk4\Ui\Label::addTo($app, ['Callback URL:']);
$cb = \Atk4\Ui\JsCallback::addTo($label);

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

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

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

```
{
    "success": true,
    "atkjs": "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 can be passed to *View::on()* as a JS 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:

```
$label = \Atk4\Ui\Label::addTo($app, ['Callback URL:']);
$cb = \Atk4\Ui\JsCallback::addTo($label);

$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:

```
$label = \Atk4\Ui\Label::addTo($app, ['Callback test']);

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

3.8.6.1 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 Atk4\Ui\JsCallback::$**confirm**

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

```
$label = \Atk4\Ui\Label::addTo($app, ['Callback URL:']);
$cb = \Atk4\Ui\JsCallback::addTo($label);

$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:

```
$label = \Atk4\Ui\Label::addTo($app, ['Callback test']);

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

3.8.6.2 JavaScript arguments

Atk4\Ui\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:

```
$label = \Atk4\Ui\Label::addTo($app);
$cb = \Atk4\Ui\JsCallback::addTo($label);

$cb->set(function (\Atk4\Ui\Js\Jquery $j, int $windowWidth) {
    return 'width is ' . $windowWidth;
}, [
    new \Atk4\Ui\Js\JsCallbackLoadableValue(new JsExpression('$(window).width()'),␣
↪static fn ($v) => (int) $v),
]);

$label->detail = $cb->getUrl();
$label->on('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 $windowWidth inside my callback. The *View::on()* also supports argument passing:

```
$label = \Atk4\Ui\Label::addTo($app, ['Callback test']);

$label->on('click', function (Jquery $j, int $windowWidth) {
    return 'width is ' . $windowWidth;
}, ['confirm' => 'sure?', 'args' => [
    new \Atk4\Ui\Js\JsCallbackLoadableValue(new JsExpression('$(window).width()'),␣
↪static fn ($v) => (int) $v),
]]);
```

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

```
$label = \Atk4\Ui\Label::addTo($app, ['Callback test']);

$label->on('click', function (Jquery $j, int $windowWidth) {
    return 'width is ' . $windowWidth;
}, [
    new \Atk4\Ui\Js\JsCallbackLoadableValue(new JsExpression('$(window).width()'),␣
↪static fn ($v) => (int) $v),
]);
```

3.8.6.3 Referring 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 like:

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

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()*.

3.9 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*:

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

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

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:

3.9.1 VirtualPage Introduction

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

class Atk4\Ui**VirtualPage**

Unlike any of the Callback classes, VirtualPage is a legitimate *View*, but it's behavior 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:

```
$vp = \Atk4\Ui\VirtualPage::addTo($layout);
\Atk4\Ui\LoremIpsum::addTo($vp);
```

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 callback for generating dynamic content for the tab:

```
$tabs->addTab('Dynamic Tab Content', function (VirtualPage $vp) {
    \Atk4\Ui\LoremIpsum::addTo($vp);
});
```

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 Atk4\Ui\VirtualPage::$**cb**

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

```
$vp = \Atk4\Ui\VirtualPage::addTo($layout);
\Atk4\Ui\LoremIpsum::addTo($vp);

$label = \Atk4\Ui\Label::addTo($layout);

$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.

property Atk4\Ui\VirtualPage::$**urlTrigger**

See *Callback::$urlTrigger*.

3.9.1.1 Output Modes

`Atk4\Ui\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 experiment with:

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

3.9.1.2 Setting Callback

`Atk4\Ui\VirtualPage::set(`*$callback*`)`

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

```
$vp = \Atk4\Ui\VirtualPage::addTo($layout);
$vp->set(function (\Atk4\Ui\VirtualPage $vp) {
    \Atk4\Ui\LoremIpsum::addTo($vp);
});

$label = \Atk4\Ui\Label::addTo($layout);

$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:

```
$tabs = \Atk4\Ui\Tabs::addTo($layout);

\Atk4\Ui\LoremIpsum::addTo($tabs->addTab('Tab1')); // regular tab
$tabs->addTab('Tab2', function (VirtualPage $p) { // dynamic tab
    \Atk4\Ui\LoremIpsum::addTo($p);
});
```

`Atk4\Ui\VirtualPage::getUrl(`*$mode*`)`

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

property `Atk4\Ui\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:

```
$vp = \Atk4\Ui\VirtualPage::addTo($layout);
\Atk4\Ui\LoremIpsum::addTo($vp);
$vp->ui = 'red inverted segment';

$label = \Atk4\Ui\Label::addTo($layout);

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

3.9.2 Loader

class Atk4\Ui**Loader**

Atk4\Ui\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:

```
$loader = \Atk4\Ui\Loader::addTo($app);
$loader->set(function (\Atk4\Ui\Loader $p) {
    // simulate slow-loading component
    sleep(2);
    \Atk4\Ui\LoremIpsum::addTo($p);
});
```

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 Atk4\Ui\Loader::$**shim**

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

```
$loader = \Atk4\Ui\Loader::addTo($app, ['shim' => [\Atk4\Ui\Message::class, 'Please wait␣
↪until we load LoremIpsum...', 'class.red' => true]]);
$loader->set(function (\Atk4\Ui\Loader $p) {
    // simulate slow-loading component
    sleep(2);
    \Atk4\Ui\LoremIpsum::addTo($p);
});
```

3.9.2.1 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.

`Atk4\Ui\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

property `Atk4\Ui\Loader::$loadEvent`

If you have NOT invoked jsLoad in your code, Loader will automatically assign it do DOM Ready(). If the automatic behavior 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:

```
$loader = \Atk4\Ui\Loader::addTo($app, ['loadEvent' => 'kaboom']);

$loader->set(function (\Atk4\Ui\Loader $p) {
    \Atk4\Ui\LoremIpsum::addTo($p);
});

\Atk4\Ui\Button::addTo($app, ['Load data'])
    ->on('click', $loader->js()->trigger('kaboom'));
```

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

3.9.2.2 Reloading

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

3.9.2.3 Inline Editing Example

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

```
$box = \Atk4\Ui\View::addTo($app, ['ui' => 'segment']);

$loader = \Atk4\Ui\Loader::addTo($box, ['loadEvent' => 'edit']);
\Atk4\Ui\Table::addTo($loader)
    ->setModel($data)
    ->addCondition('year', $app->stickyGet('year'));
```

(continues on next page)

(continued from previous page)

```
\Atk4\Ui\Button::addTo($box, ['Edit Data Settings'])
    ->on('click', $loader->js()->trigger('edit'));

$loader->set(function (\Atk4\Ui\Loader $p) {
    $form = \Atk4\Ui\Form::addTo($p);
    $form->addControl('year');

    $form->onSubmit(function (Form $form) use ($p) {
        return new \Atk4\Ui\Js\JsReload($p, ['year' => $form->entity->get('year')]);
    });
});
```

3.9.2.4 Progress Bar

property Atk4\Ui\Loader::$**progressBar**

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:

```
$loader = \Atk4\Ui\Loader::addTo($app, ['progressBar' => true]);
$loader->set(function (\Atk4\Ui\Loader $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);

    \Atk4\Ui\LoremIpsum::addTo($p);
});
```

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.

3.10 Documentation is coming soon.

CHAPTER

FOUR

FILE STRUCTURE EXAMPLE & FIRST APP

We will deal here with a suggestion how you could structure your files and folders for your individual atk4 project. Let's assume you are planning to create at least several pages with some models and views. This example can be expanded and modified to your needs and shows just one concept of how to setup an atk4 project.

4.1 File structure example

This file structure is a recommendation and no must. It is a best practice example. Feel free to experiment with it and find the ideal file structure for your project.

- config
  - db.php
- public_html
  - images
  - index.php
  - init.php
  - admin.php (if needed)
- projectfolder (could be named "app" for example)
  - Forms
    - ExampleForm1.php
    - ExampleForm2.php
    - UserDetailForm.php
  - Models
    - ExampleClass1.php
    - ExampleClass2.php
    - LoadAllUsers.php
  - Views
    - View1.php
    - View2.php
    - GridUserList.php
- vendor (contains all needed composer modules - don't touch them)

  - autoload.php
  - ...
- composer.json

4.2 Composer configuration

Configure your composer.json to load the atk4 AND your project folder. Your composer.json could look like this:

```
{
    "require":{
        "atk4/ui": "*"
    },
    "autoload": {
        "psr-4": {
            "MyProject\\": "projectfolder/"
        }
    }
}
```

4.2.1 What does that mean?

As soon as you start a "composer update" or "composer dump-autoload" in the public_html directory, all needed atk4 files and all your project files from the subdirectory "projectfolder" are processed by Composer and the autoload.php file is generated. Read below how to load the autoload.php into your project.

The "require" section within composer.json loads publicly available composer packages from the server.

The "autoload" section within composer.json loads your individual project files (that are saved locally on your computer). "MyProject" defines the namespace you are using in your classes later on.

4.2.2 Why "public_html"?

It is a good idea to keep away sensible configuration files that contain passwords (like database connection setups etc.) from the public and make it only available to your application. If you go live with your app, you load everything up to the webspace (config & public_html) and point the domain to the public_html directory.

If you call www.myexampledomain.com it should show the content of public_html. Within a php file from public_html you are still able to access and include files from config. But you can't call it directly through the domain (that means in our case "db.php" can't be accessed through the domain).

4.3 Create your application

To initialize your application we need to do the following steps:

1. Create db.php for database
2. Create init.php
3. Load Composer autoload.php (which loads up atk4) in init.php
4. Initialize the app class in init.php

5. Create index.php and admin.php

4.3.1 Create db.php for database

We initialize a reusable database connection in db.php through a mysql persistence. Create a file called “db.php” in the directory “config”:

```
<?php
$db = \Atk4\Data\Persistence::connect("mysql://localhost/#myexampledatabase", "#username
↪", "#password");
```

Please remember to use a database that still exists.

4.3.2 Create init.php, index.php and / or admin.php files

Create a new file in “public_html/projectfolder” and name it “init.php”. In this file we load up our app (later) and load the database configuration:

```
<?php
$rootdir = "../"; // the public_html directory
require_once $rootdir . "../config/db.php"; // contains database configuration outside␣
↪the public_html directory
```

4.3.3 Load Composer autoload.php (which loads up atk4) in init.php

```
require_once $rootdir . "vendor/autoload.php"; // loads up atk4 and our project files␣
↪from Composer
```

4.3.4 Initialize the app class in init.php

```
$app = new \Atk4\Ui\App(['title' => 'Welcome to my first app']); // initialization of␣
↪our app
$app->db = $db; // defines our database for reuse in other classes
```

4.3.5 Create index.php and admin.php

If you want to write an app with a backend, create a file called “admin.php”:

```
<?php
$rootdir = "../";
require_once __DIR__ . "init.php";
$app->initLayout([\Atk4\Ui\Layout\Admin::class]);
```

If you want to write an app with a frontend, create a file called “index.php”:

```
<?php
$rootdir = "../";
require_once __DIR__ . "init.php";
$app->initLayout([\Atk4\Ui\Layout\Centered::class]);
```

4.4 Create your own classes

Now as your basic app is set up and running, we start implementing our own classes that build the core of our app. Following the PSR-4 specifications all class names and file names have to correspond to each other.

If we want to create a class called “myFirstClass” we have to save it to a file called “myFirstClass.php”.

Let’s do our first class. Please create a new file in the directory “projectfolder/Views” and call it “View1.php”.

Now comes a tricky part: you have to define a namespace within your class file that corresponds with the namespace you have defined in your composer.json. Do you remember? - If no, take a look at the beginning of this document. We defined there “MyProject” as our namespace for the directory “projectfolder”.

Open the created file “View1.php” in your editor and add the following lines:

```
<?php
namespace MyProject\Views;

class View1 extends \Atk4\Data\View
{
    protected function init(): void
    {
        parent::init();

        \Atk4\Ui\Text::addTo($this->getApp(), ['here goes some text']);
    }
}
```

“namespace MyProject\Views;” defines the namespace to use. It reflects the folder structure of the app. The file located in “projectfolder/Views/View1.php” becomes “MyProject\Views\View1” in the namespace.

For each of your classes create a separate file. As long as you follow the name conventions all your class files will be autoloaded by Composer.

> **Warning:** Keep in mind that as soon as you have created one or more new file(s) within the projectfolder you have to run “composer dump-autoload”!!! Otherwise the newly generated file(s) and classes will not be autoloaded and are therefore unavailable in your application.

4.5 Load your class in index.php

To use our class in our app, we have to include it into our app. This can be done either through index.php or admin.php.

Please add the following lines into your index.php:

```
\MyProject\Views\View1::addTo($app);
```

or if you have added at the beginning of your index.php "use MyProject\Views\View1;" you can write:

```
View1::addTo($app);
```

See also *Using namespaces* on this topic...

CHAPTER

FIVE

COMPONENTS

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

5.1 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.

5.1.1 Views

Agile UI is a component framework, which follows a software patterns known as "render tree" and "two pass HTML rendering".

class Atk4\Ui**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:

```
$v = new \Atk4\Ui\View(['ui' => 'segment', 'class.inverted' => true]);
Button::addTo($v, ['Orange', 'class.inverted orange' => true]);
```

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 render tree `Views` you need to execute `renderToHtml()` for the top-most leaf:

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

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

5.1.1.1 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 actuallized.

`Atk4\Ui\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->getApp() into $object->getApp().

If this object is initialized, will also initialize $object

Parameters

- **$object** – Object or *Seed* to add into render tree.
- **$region** – When outputting HTML, which region in *View::$template* to use.

`Atk4\Ui\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', 'class.red' => true]);
$top = new \Atk4\Ui\View(['ui' => 'segments']);
$bottom = new \Atk4\Ui\Button(['Hello World', 'class.orange' => true]);

// not arranged into render-tree yet

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

// still not sure if finished adding

$app = new \Atk4\Ui\App(['title' => 'My App']);
$app->initLayout($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(['title' => 'My App']);
$top = $app->initLayout(new \Atk4\Ui\View(['ui' => 'segments']));

$middle = View::addTo($top, ['ui' => 'segment', 'class.red' => true]);

$bottom = Button::addTo($middle, ['Hello World', 'class.orange' => true]);
```

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

```
$app = new \Atk4\Ui\App(['title' => 'My App']);
$top = $app->initLayout([\Atk4\Ui\View::class, 'ui' => 'segments']);

$middle = View::addTo($top, ['ui' => 'segment', 'class.red' => true]);

$bottom = Button::addTo($middle, ['Hello World', 'class.orange' => true]);
```

The rest of documentation will use this concise 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 `Factory::factory()`, which will be responsible of instantiating the actual object.

(TODO: link to App:Factory)

5.1.1.2 Use of $app property and Dependency Injeciton

property Atk4\Ui\View::**$app**

Each View has a property $app that is defined through \Atk4\Core\AppScopeTrait. View elements rely on persistence of the app class in order to perform Dependency Injection.

Consider the following example:

```
$app->logger = new Logger('log'); // Monolog

// next, somewhere in a render tree
$view->getApp()->logger->log('Foo Bar');
```

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

5.1.1.3 Integration with Agile Data

Atk4\Ui\View::**setModel**(*$model*)

Associate current view with a domain model.

property Atk4\Ui\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:

```
$db = new \Atk4\Data\Persistence\Sql($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:

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

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

```
$app->db = new \Atk4\Data\Persistence\Sql($dsn);

// next, anywhere in a view
```

(continues on next page)

(continued from previous page)

```
$client = new Client($this->getApp()->db);
$form->setEntity($client);
```

Or if you prefer a more consise code:

```
$app->db = new \Atk4\Data\Persistence\Sql($dsn);

// next, anywhere in a view
$form->setEntity('Client');
```

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

5.1.1.4 UI Role and Classes

`Atk4\Ui\View::__construct`(*$defaults = []*)

Parameters

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

property `Atk4\Ui\View::$ui`

Indicates a role of a view for CSS framework.

A constructor of a view often maps into a 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:

```
$view = View::addTo($app, ['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:

```
$view = View::addTo($app, ['ui' => 'segment', 'name' => 'test-id']);
```

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

```
$view = View::addTo($app, ['ui' => 'segment']);
$view->name = '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:

```
$view = View::addTo($app, ['class.orange' => true, 'ui' => 'segment']);
$view->name = 'test-id';
```

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

property `Atk4\Ui\View::$class`

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

Atk4\Ui\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

Atk4\Ui\View::**removeClass**(*$class*)

Parameters

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

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

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

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

5.1.1.5 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 \Atk4\Ui\Icon() inside the button.

The same pattern can be used for other scenarios:

```
$button = Button::addTo($app, ['icon' => 'book']);
```

This code will have same effect as:

```
$button = Button::addTo($app);
$button->icon = 'book';
```

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

```
Icon::addTo($button, ['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:

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

5.1.1.6 Rendering of a Tree

Atk4\Ui\View::**renderToHtml**()

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.
- return JS code with on-dom-ready instructions along with HTML code of a current view.

You should not override `renderToHtml()` in your objects. If you are integrating Agile UI into your framework you shouldn't even use `renderToHtml()`, but instead use `getHtml` and `getJs`.

`Atk4\Ui\View::getHtml()`

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

`Atk4\Ui\View::getJs()`

Returns JsBlock containing JS chains that were assigned to current element or it's children.

5.1.1.7 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.

`Atk4\Ui\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():

```
protected function renderView(): void
{
    if (str_len($this->info) > 100) {
        $this->addClass('tiny');
    }

    parent::renderView();
}
```

It's important when you call parent. You wouldn't be able to affect template of a current view anymore after calling renderView.

Also, note that child classes are rendered already before invocation of rederView. If you wish to do something before child render, override method *View::recursiveRender()*

property `Atk4\Ui\View::$template`

Template of a current view. This attribute contains an object of a class `HtmlTemplate`. You may secify this value explicitly:

```
View::addTo($app, ['template' => new \Atk4\Ui\HtmlTemplate('<b>hello</b>')]);
```

property `Atk4\Ui\View::$defaultTemplate`

By default, if value of *View::$template* is not set, then it is loaded from class specified in `defaultTemplate`:

```
View::addTo($app, ['defaultTemplate' => './mytpl.html']);
```

You should specify defaultTemplate using relative path to your project root or, for add-ons, relative to a current file:

```
// in Add-on
View::addTo($app, ['defaultTemplate' => __DIR__ . '/../templates/mytpl.htmtl']);
```

Agile UI does not currently provide advanced search path for templates, by default the template is loaded from folder `vendor/atk4/ui/template`. To change this behavior, see *App::loadTemplate()*.

property Atk4\Ui\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';

    protected function renderView(): void
    {
        parent::renderView();

        $this->template->set('title', $this->title);
    }
}
```

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

```
MyView::addTo($app, ['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:

```
$profile = View::addTo($app, ['template' => 'myview.html']);
$profile->setEntity($user);
Lister::addTo($profile, [], ['Tags'])->setModel($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 `HtmlTemplate`.

5.1.1.8 Unique ID tag

property Atk4\Ui\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 'name' property:

```
$b = new \Atk4\Ui\Button(['name' => 'my-button3']);
echo $b->renderToHtml();
```

Outputs:

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

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

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

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

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

property Atk4\Ui\View::**$name**

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

5.1.1.9 Reloading a View

Atk4\Ui\View::**jsReload**(*$getArgs*)

Agile UI makes it easy to reload any View on the page. Starting with v1.4 you can now use View::JsReload(), which will respond with JavaScript Action for reloading the view:

```
$b1 = Button::addTo($app, ['Click me']);
$b2 = Button::addTo($app, ['Rand: ' . rand(1, 100)]);

$b1->on('click', $b2->jsReload());

// previously:
// $b1->on('click', new \Atk4\Ui\Js\JsReload($b2));
```

5.1.1.10 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->setStyle('align', 'right');
$view->setAttr('href', 'https://...');
```

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

property Atk4\Ui\View::$**content**

Set static contents of this view.

Atk4\Ui\View::**setProperties**(*$properties*)

Parameters

- **$properties** –

Atk4\Ui\View::**setProperty**(*$key*, *$val*)

Parameters

- **$key** –
- **$val** –

Atk4\Ui\View::**set**(*$arg1 = []*, *$arg2 = null*)

Parameters

- **$arg1** –
- **$arg2** –

Atk4\Ui\View::**recursiveRender**()

5.1.2 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.

5.1.2.1 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:

```
$view = View::addTo($app, ['template' => 'myview.html']);
```

Now your application should contain list of 3 sample countries 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, because View clears out all extra template tags.

Next I'll add Lister:

```
Lister::addTo($view, [], ['Countries'])
    ->setModel(new Country($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>
```

5.1.2.2 Tweaking the output

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

```
$lister->onHook(\Atk4\Ui\Lister::HOOK_BEFORE_ROW, function (\Atk4\Ui\Lister $lister) {
    $lister->currentRow->set('iso', mb_strtolower($lister->currentRow->get('iso')));
});
```

5.1.2.3 Model vs Static Source

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

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

5.1.2.4 Special template tags

Your {row} template 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::$titleField)
- {$_href} - will point to current page but with ?id=123 extra GET argument.

5.1.2.5 Load page content dynamically when scrolling

You can make lister load page content dynamically when user is scrolling down page.

```
$lister->addJsPaginator(20, $options = [], $container = null, $scrollRegion = null);
```

The first parameter is the number of item you wish to load per page. The second parameter is options you want to pass to respective JS widget. The third parameter is the $container view holding the lister and where scrolling is applicable. And last parameter is CSS selector of element in which you want to do scrolling.

5.1.2.6 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:

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

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

5.1.3 Table

class Atk4\Ui**Table**

Important: For columns, see *Table\Column*. For DIV-based lists, see *Lister*. For an interactive features see *Grid* and *Crud*.

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://fomantic-ui.com/collections/table.html.
- Support for data decorating. (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

5.1.3.1 Basic Usage

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

```
$table = Table::addTo($app);
$table->setModel(new Order($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:

```
$myArray = [
    ['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')],
];

$table = Table::addTo($app);
$table->setSource($myArray);

$table->addColumn('name');
$table->addColumn('surname', [\Atk4\Ui\Table\Column\Link::class, 'url' => 'details.php?
↪surname={$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`**(*\Atk4\Data\Model $model*, *$fields = null*)

`Atk4\Ui\Table::`**`addColumn`**(*$name*, *$columnDecorator = []*, *$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:

```
$table = Table::addTo($app);
$table->setModel(new Order($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), []); // [] 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::addControl*.

5.1.3.2 Calculations

Apart from adding columns that reflect current 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 https://atk4-data.readthedocs.io/en/develop/expressions.html)

It's always a good idea to calculate column inside database. 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 already defined in `Order::init()`:

```
$table = Table::addTo($app);
$order = new Order($db);

$order->addExpression('total', '[price] * [amount]')->type = 'atk4_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 'atk4_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 uiPersistence.

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:

```
$table = Table::addTo($app);
$order = new Order($db);

$order->addExpression('total', [
    '[price]*[amount]',
    'type' => 'atk4_money',
    'caption' => 'Total Price',
]);

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

Column Objects

To read more about column objects, see *Table Column Decorators*

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(\Atk4\Data\Field $field)`

If the value of the field can be displayed by *Table\Column* 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::$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 columns without field:

```
$action = $this->addColumn(null, [Table\Column\ActionButtons::class]);
$action->addButton('Delete', 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 *Table\Column::getHeaderCellHtml*, *Table\Column::getTotalsCellHtml* and *Table\Column::getDataCellHtml*.

Some columns require name, such as *Table\Column* 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:

```
$table = Table::addTo($app);
$order = new Order($db);

$table->setModel($order, ['name', 'price', 'amount', 'status']);
$table->addColumn('total', new \Atk4\Data\Field\Calculated(function (Model $row) {
    return $row->get('price') * $row->get('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():

```
$table = Table::addTo($app);
$order = new Order($db);
```

(continues on next page)

(continued from previous page)

```
$table->setModel($order, ['name', 'price', 'amount']);
$table->addColumn('total', new \Atk4\Data\Field\Calculated(function (Model $row) {
    return $row->get('price') * $row->get('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.

`Atk4\Ui\Table::`**`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', $fieldDef], ['delete', $deleteColumn]);

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

```
$colGap = new \Atk4\Ui\Table\Column\Template('<td> ... <td>');

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

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

5.1.3.3 Table sorting

property `Atk4\Ui\Table::$`**`sortable`**

property `Atk4\Ui\Table::$`**`sortBy`**

property `Atk4\Ui\Table::$`**`sortDirection`**

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

```
$table->sortable = true;
$table->sortBy = 'name';
$table->sortDirection = 'asc';
```

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->getApp()->includeJS('https://fomantic-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 *Table\Column::getHtmlTags*:

```
class ExpiredColumn extends \Atk4\Ui\Table\Column
    public function getDataCellHtml(): string
    {
        return '{$_expired}';
    }

    public function getHtmlTags(\Atk4\Data\Model $row, ?\Atk4\Data\Field $field): array
    {
        return [
            '_expired' => $field->get($row) < 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::$useHtmlTags` property is disabled (for performance).

5.1.3.4 Table 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
$dropbox = \Atk4\Dropbox\Persistence($dbConfig);
$files = new \Atk4\Dropbox\Model\File($dropbox);

Table::addTo($app)->setModel($files);

// show contents of dropbox
$github = \Atk4\Github\IssuePersistence($githubApiConfig);
$issues = new \Atk4\Github\Model\Issue($github);
```

(continues on next page)

(continued from previous page)

```
Table::addTo($app)->setModel($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 any time 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 "decorators".

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

1. Generate header row.
2. Iterate through rows
    1. Current row data is accessible through $table->model property.
    2. Generate template for data row.
    3. Update Totals if `Table::addTotals` was used.
    4. Insert row values into `Table::$tRow`
        1. Template relies on uiPersistence for formatting values
    5. Collect HTML tags from 'getHtmlTags' hook.
    6. Collect getHtmlTags() from columns objects
    7. Inject HTML into `Table::$tRow` template
    8. Render and append row template to Table Body ({$Body})
3. If no rows were displayed, then "empty message" will be shown (see `Table::$tEmpty`).
4. If `addTotals` was used, append totals row to table footer.

5.1.3.5 Dealing with Multiple decorators

`Atk4\Ui\Table::`**`addDecorator`**`(`*$name*, *$columnDecorator*`)`

`Atk4\Ui\Table::`**`getColumnDecorators`**`(`*$name*`)`

Decorator is an object, responsible for wrapping column data into a table cell (td/tr). This object is also responsible for setting class of the column, labeling the column and somehow making it look nicer especially inside a table.

Important: Decorating is not formatting. If we talk "date", then in order to display it to the user, date must be in a proper format. Formatting of data is done by `Persistence\Ui` and is not limited to the table columns. Decorators may add an icon, change cell style, align cell or hide overflowing text to make table output look better.

One column may have several decorators:

```
$table->addColumn('salary', new \Atk4\Ui\Table\Column\Money());
$table->addDecorator('salary', new \Atk4\Ui\Table\Column\Link(['page2']));
```

In this case the first decorator will take care of tr/td tags but second decorator will compliment it. Result is that table will output 'salary' as a currency (align and red ink) and also decorate it with a link. The first decorator will be responsible for the table column header. If field type is not set or type is like "integer", then a generic formatter is used.

There are a few things to note:

1. Property *Table::$columns* contains either a single or multiple decorators for each column. Some tasks will be done by first decorator only, such as getting TH/header cell. Others will be done by all decorators, such as collecting classes / styles for the cell or wrapping formatted content (link, icon, template).
2. formatting is always applied in same order as defined - in example above Money first, Link after.
3. output of the \Atk4\Ui\Table\Column\Money decorator is used into Link decorator as if it would be value of cell, however decorators have access to original value also. Decorator implementation is usually aware of combinations.

`Table\Column\Money::getDataCellTemplate` is called, which returns ONLY the HTML value, without the <td> cell itself. Subsequently `Table\Column\Link::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 concatenated with other table columns just before rendering starts. The actual template is formed by calling. This may be too much detail, so if you need to make a note on how template caching works then,

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

Header and Footer

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

Redefining

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

5.1.3.6 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*

JsPaginator

`Atk4\Ui\Table::`**`addJsPaginator`**(*$ipp*, *$options = []*, *$container = null*, *$scrollRegion = 'Body'*)

JsPaginator will load table content dynamically when user scroll down the table window on screen.

```
$table->addJsPaginator(30);
```

See also `Lister::addJsPaginator`

Resizable Columns

`Atk4\Ui\Table::`**`resizableColumn`**(*$fx = null*, *$widths = null*, *$resizerOptions = []*)

Each table's column width can be resize by dragging the column right border:

```
$table->resizableColumn();
```

You may specify a callback function to the method. The callback will return an array containing each column name in table with their new width in pixel.:

```
$table->resizableColumn(function (Jquery $j, array $columnWidths) {
    // do something with new column widths
}, [200, 300, 100, 100, 100]);
```

Note that you may specify an array of integer representing the initial width value in pixel for each column in your table.

Finally you may also specify some of the resizer options - https://github.com/Bayer-Group/column-resizer#options

5.1.3.7 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(
    'window.location = \'page.php?id=\' + []',
    [(new Jquery())->closest('tr')->data('id')]
));
```

See also *JavaScript Mapping*.

Static Attributes and classes

class Atk4\Ui**Table\Column**

Atk4\Ui\Table\Column::**addClass**(*$class*, *$scope = 'body'*)

Atk4\Ui\Table\Column::**setAttr**(*$attribute*, *$value*, *$scope = 'body'*)

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

```
$table->addColumn('name', (new \Atk4\Ui\Table\Column()->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\Table\Column()->addClass('right aligned', 'all
↪')));
```

There are several ways to make your code more readable:

```
$table->addColumn('name', new \Atk4\Ui\Table\Column())
    ->addClass('right aligned', 'all');
```

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

```
$table->addColumn('name', [\Atk4\Ui\Table\Column::class])
    ->addClass('right aligned', 'all');
```

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

```
$table->addColumn('name', [\Atk4\Ui\Table\Column::class])
    ->setAttr('colspan', 2, 'all');
```

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

Please note that if you are redefining *Table\Column::getHeaderCellHtml*, *Table\Column::getTotalsCellHtml* or *Table\Column::getDataCellHtml* and you wish to preserve functionality of setting custom attributes and classes, you should generate your TD/TH tag through getTag method.

Atk4\Ui\Table\Column::**getTag**(*$position*, *$attr*, *$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:

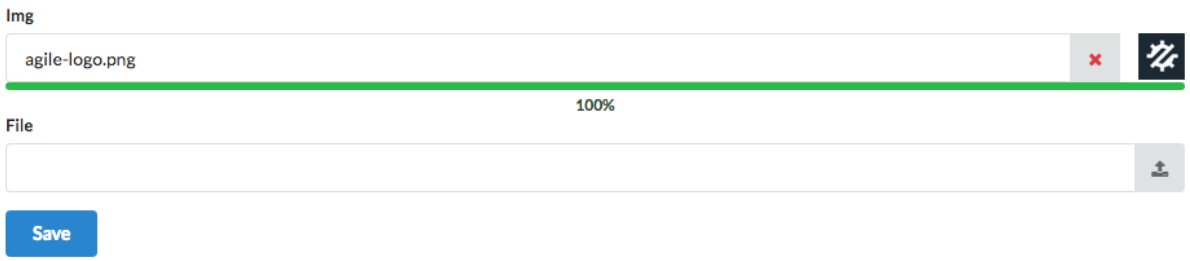
```
$cb = $table->addColumn('CheckBox');
```

Using dynamic values

Body attributes will be embedded into the template by the default *Table\Column::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 *Table\Column\Status*.

5.1.4 File Upload



Upload (and UploadImage) classes implement form controls that can be used to upload files or images. Implementation of *Form* in Agile UI submits data using JavaScript request and therefore files should be uploaded before form submission. Process used can be described in steps:

1. User arrives at the page with a form
2. User selects file.
3. File begins uploading.
4. PHP upload callback `Form\Control\Upload::onUpload` is called, returns "file_id"
5. "file_id" is placed inside form.
6. User submits the form
7. *Form::onSubmit()* receives "file_id"

Currently only one file can be uploaded at a time. If file is uploaded incorrectly, it can be removed. Both Upload and UploadImage controls contain an upload button which would open a File Selection dialog. UploadImage also implements image preview icon. During upload, a progress bar will appear.

class Atk4\Ui**Form\Control\Upload**

5.1.4.1 Attributes

Upload control has the following properties:

property Atk4\Ui\Form\Control\Upload::**$accept**

An array of string containing the file type accepted by the form control, default is empty. Example would be: `['application/pdf', 'images/*']`.

property Atk4\Ui\Form\Control\Upload::**$action**

The button view to use for displaying the file open dialog. A default action button is used if omitted.

5.1.4.2 Callbacks

When adding an Upload or UploadImage field to a form, onUpload and onDelete callback must be defined:

```
$img = $form->addControl('img', [\Atk4\Ui\Form\Control\UploadImage::class, ['defaultSrc'␣
↪=> './images/default.png', 'placeholder' => 'Click to add an image.']]);

$img->onUpload(function (array $postFile) {
    // callback action here...
});

$img->onDelete(function (string $fileId) {
    // callback action here...
});
```

onUpload

The onUpload callback get called as soon as the upload process is finished. This callback function will receive the `$_FILES['upfile']` array as function parameter (see https://php.net/manual/en/features.file-upload.php).

The onUpload callback function is a good place to:

- ensure the file is of a proper type and safe,
- move file to a proper location on server or in a cloud,
- save file property in db,
- setup a fileId that will used on a form form save,
- setup a file preview to display back to user,
- notify your user of the file upload process,

Example showing the onUpload callback on the UploadImage field:

```
$img->onUpload(function (array $postFile) use ($form, $img) {
    if ($postFile['error'] !== 0) {
        return $form->jsError('img', 'Error uploading image.');
    }

    // do file processing here...

    $img->setThumbnailSrc('./images/' . $fileName);
```

(continues on next page)

(continued from previous page)

```
    $img->setFileId('123456');

    // can also return a notifier
    return new \Atk4\Ui\Js\JsToast([
        'message' => 'File is uploaded!',
        'class' => 'success',
    ]);
});
```

When user submit the form, the form control data value that will be submitted is the fileId set during the onUpload callback. The fileId is set to file name by default if omitted:

```
$form->onSubmit(function (Form $form) {
    // implement submission here
    return $form->jsSuccess('Thanks for submitting file: ' . $form->entity->get('img'));
});
```

onDelete

The onDelete callback get called when user click the delete button. This callback function receive the same fileId set during the onUpload callback as function parameter.

The onDelete callback function is a good place to:

- validate ID (as it can technically be changed through browser's inspector)
- load file property from db
- remove previously uploaded file from server or cloud,
- delete db entry according to the fileId,
- reset thumbnail preview,

Example showing the onDelete callback on the UploadImage field:

```
$img->onDelete(function (string $fileId) use ($img) {
    // reset thumbanil
    $img->clearThumbnail('./images/default.png');

    return new \Atk4\Ui\Js\JsToast([
        'message' => $fileId . ' has been removed!',
        'class' => 'success',
    ]);
});
```

5.1.4.3 UploadImage

Similar to Upload, this is a control implementation for uploading images. Here are additional properties:

class Atk4\Ui**Form\Control\UploadImage**

UploadImage form control inherits all of the Upload properties plus these ones:

property Atk4\Ui\Form\Control\UploadImage::$**thumbnail**

The thumbnail view associated with the form control.

property Atk4\Ui\Form\Control\UploadImage::$**thumbnailRegion**

The region in input template where to add the thumbnail view, default to AfterAfterInput region.

property Atk4\Ui\Form\Control\UploadImage::$**defaultSrc**

The default image source to display to user, prior to uploading the images.

5.1.5 Table Column Decorators

Classes like *Table* and Card do not render their cell contents themselves. Instead they rely on Column Decorator class to position content within the cell.

This is in contrast to *View* and *Lister* which do not use Table/Cell and therefore Column decorator is not required.

All column decorators in Agile UI have a base class *Table\Column*. Decorators will often look at the content of the associated value, for example Money will add cell class negative only if monetary value is less than zero. The value is taken from Model's Field object.

Column decorators can also function without associated value. *Template* may have no fields or perhaps display multiple field values. Action displays interactive buttons in the table. CheckBox makes grid rows selectable. Ordering displays a draggable handle for re-ordering rows within the table.

A final mention is about Multiformat, which is a column decorator that can swap-in any other decorator based on condition. This allows you to change button [Archive] for active records, but if record is already archived, use a template "Archived on {$archive_date}".

5.1.5.1 Generic Column Decorator

class Atk4\Ui**Table\Column**

Generic description of a column for *Table*

Table object relies on a separate class: \Atk4\Ui\Table\Column to present most of the values. The goals of the column object is to format anything around the actual values. The type = 'atk4_money' will result in a custom formatting of the value, but will also require column to be right-aligned. To simplify this, type = 'atk4_money' will use a different column class - *Table\Column\Money*. There are several others, but first we need to look at the generic column and understand it's base capabilities:

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

Atk4\Ui\Table\Column::**getHeaderCellHtml**(*\Atk4\Data\Field $field*) → string

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

`Atk4\Ui\Table\Column::getTotalsCellHtml(\Atk4\Data\Field $field, $value)` → string

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

`Atk4\Ui\Table\Column::getDataCellHtml(\Atk4\Data\Field $field)` → string

Provided with a field, this method will respond with HTML **template**. When iterating, a combined template will be used to display the values.

A sample template could be:

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

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

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

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

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

`Atk4\Ui\Table\Column::getHtmlTags($model, $field = null)`

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

5.1.5.2 Column Menus and Popups

Table column may have a menu as seen in https://ui.atk4.org/demos/collection/tablecolumnmenu.php. Menu is added into table column and can be linked with Popup or Menu.

Basic Use

The simplest way to use Menus and Popups is through a wrappers: *Grid::addDropdown* and *Grid::addPopup*:

```
View::addTo($grid->addPopup('iso'))
    ->set('Grid column popup text');

// OR

$grid->addDropdown('name', ['Sort A-Z', 'Sort by Relevance'], function (string $item) {
    return $item;
});
```

Those wrappers will invoke methods *Table\Column::addDropdown* and *Table\Column::addPopup* for a specified column, which are documented below.

Popups

Atk4\Ui\Table\Column::**addPopup**()

To create a popup, you need to get the column decorator object. This must be the first decorator, which is responsible for rendering of the TH box. If you are adding column manually, *Table::addColumn()* will return it. When using model, use *Table::getColumnDecorators*:

```
$table = Table::addTo($app, ['class.celled' => true]);
$table->setModel(new Country($app->db));

$nameColumn = $table->getColumnDecorators('name');
LoremIpsum::addTo($nameColumn[0]->addPopup());
```

Important: If content of a pop-up is too large, it may not be possible to display it on-screen. Watch for warning.

You may also use *Popup::set* method to dynamically load the content:

```
$table = Table::addTo($app, ['class.celled' => true]);
$table->setModel(new Country($app->db));

$nameColumn = $table->getColumnDecorators('name');
$nameColumn[0]->addPopup()->set(function (View $p) {
    HelloWorld::addTo($p);
});
```

Dropdown Menus

Atk4\Ui\Table\Column::**addDropdown**()

Menus will show item selection and will trigger a callback when user selects one of them:

```
$someColumn->addDropdown(['Change', 'Reorder', 'Update'], function (string $item) {
    return 'Title item: ' . $item;
});
```

5.1.5.3 Decorators for data types

In addition to *Table\Column*, Agile UI includes several column implementations.

Link

class Atk4\Ui**Table\Column\Link**

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:

```
$table->addColumn('name', [\Atk4\Ui\Table\Column\Link::class, 'https://google.com/?q={
↪$name}']);
```

The URL may also be specified as an array. It will be passed to App::url() which will encode arguments:

```
$table->addColumn('name', [\Atk4\Ui\Table\Column\Link::class, ['details', 'id' => 123, 'q
↪' => $anything]]);
```

In this case even if `$anything = '{$name}'` the substitution will not take place for safety reasons. To pass on some values from your model, use second argument to constructor:

```
$table->addColumn('name', [\Atk4\Ui\Table\Column\Link::class, ['details', 'id' => 123], [
↪'q' => 'name']]);
```

Money

class Atk4\Ui\Table\Column\Money

Helps decorating monetary values. Will align value to the right and if value is less than zero will also use red text (td class "negative" for Fomantic-UI). The money cells are not wrapped.

For the actual number formatting, see uiPersistence

Status

class Atk4\Ui\Table\Column\Status

Allow you to set highlight class and icon based on column value. This is most suitable for columns that contain predefined 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'],
];

$table->addColumn('status', new \Atk4\Ui\Table\Column\Status($states));
```

Current list of states supported:

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

(list of states may be expanded further)

Template

class Atk4\Ui**Table\Column\Template**

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:

```
$table->addColumn('rating', new \Atk4\Ui\Table\Column\Template('{$rating} out of 10'));
```

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

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

Image

class Atk4\Ui**Table\Column\Image**

This column is suitable if you wish to have image in your table cell:

```
$table->addColumn('image_url', new \Atk4\Ui\Table\Column\Image);
```

5.1.5.4 Interactive Decorators

ActionButtons

class Atk4\Ui**Table\Column\ActionButtons**

Can be used to add "action buttons" column to your table:

```
$action = $table->addColumn(null, [Table\Column\ActionButtons::class]);
```

If you want to have label above the action column, then:

```
$action = $table->addColumn(null, [Table\Column\ActionButtons::class, 'caption' => 'User␣
↪Actions']);
```

Atk4\Ui\Table\Column\ActionButtons::**addButton**(*$button*, *$action*, *$confirm = false*)

Adds another button into "Actions" column which will perform a certain JavaScript action when clicked. See also *Grid::addActionButton()*:

```
$button = $action->addButton('Reload Table', $table->jsReload());
```

Normally you would also want to pass the ID of the row which was clicked. You can use `Table:jsRow()` and jQuery's data() method to reference it:

```
$button = $action->addButton('Reload Table', $table->jsReload(['clicked' => $table->
↪jsRow()->data('id')]));
```

Moreover you may pass $action argument as a PHP callback.

Atk4\Ui\Table\Column\ActionButtons::**addModal**(*$button*, *$title*, *$callback*)

Triggers a modal dialog when you click on the button. See description on *Grid::addModalAction()*:

```
$action->addButton(['Say HI'], function (Jquery $j, $id) use ($g) {
    return 'Loaded "' . $g->model->load($id)->get('name') . '" from ID=' . $id;
});
```

Note that in this case ID is automatically passed to your callback.

Checkbox

class Atk4\Ui**Table\Column\Checkbox**

Atk4\Ui\Table\Column\Checkbox::**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\Table\Column\CheckBox());

$button->on('click', new JsReload($segment, ['ids' => $box->jsChecked()]));
```

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

Multiformat

Sometimes your formatting may change depending on value. For example you may want to place link only on certain rows. For this you can use an `\Atk4\Ui\Table\Column\Multiformat` decorator:

```
$table->addColumn('amount', [\Atk4\Ui\Table\Column\Multiformat::class, function (Model
↪$entity) {
    if ($entity->get('is_invoiced') > 0) {
        return [\Atk4\Ui\Table\Column\Money::class, [\Atk4\Ui\Table\Column\Link::class,
↪'invoice', ['invoice_id' => 'id']]];
    } elseif (abs($entity->get('is_refunded')) < 50) {
        return [[\Atk4\Ui\Table\Column\Template::class, 'Amount was <b>refunded</b>']];
    }

    return [[\Atk4\Ui\Table\Column\Money::class]];
}]);
```

You supply a callback to the Multiformat decorator, which will then be used to determine the actual set of decorators to be used on a given row. The example above will look at various fields of your models and will conditionally add Link on top of Money formatting.

The callback must return array of seeds like:

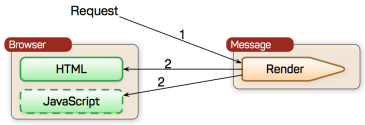
```
[[\Atk4\Ui\Table\Column\Link::class], [\Atk4\Ui\Table\Column\Money::class]]
```

Multiple decorators will be created and merged.

Note: If you are operating with large tables, code your own decorator, which would be more CPU-efficient.

5.2 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.



5.2.1 Button

class Atk4\Ui**Button**

Implements a clickable button:

```
$button = Button::addTo($app, ['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 Fomantic-UI documentation for Button to find out more about available classes: https://fomantic-ui.com/elements/button.html.

Demo: https://ui.atk4.org/demos/basic/button.php

5.2.1.1 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:

```
$button = Button::addTo($app, ['Like', 'class.blue' => true, 'icon' => 'thumbs up']);

// or

$button = Button::addTo($app, ['Like', 'class.blue' => true, '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:

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

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

5.2.1.2 Button Bar

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

```
$bar = View::addTo($app, ['ui' => 'vertical buttons']);

Button::addTo($bar, ['Play', 'icon' => 'play']);
Button::addTo($bar, ['Pause', 'icon' => 'pause']);
Button::addTo($bar, ['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);
```

5.2.1.3 Linking

`Atk4\Ui\Button::link()`

Will link button to a destination URL or page:

```
$button->link('https://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->on('click', new JsExpression('window.location.reload()'));
```

5.2.1.4 Complex Buttons

Knowledge of the Fomantic-UI button (https://fomantic-ui.com/elements/button.html) can help you in creating more complex buttons:

```
$forks = new Button(['labeled' => true]);
Icon::addTo(Button::addTo($forks, ['Forks', 'class.blue' => true]), ['fork']);
Label::addTo($forks, ['1,048', 'class.basic blue left pointing' => true]);
$app->add($forks);
```

5.2.2 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: https://fomantic-ui.com/elements/label.html.

Demo: https://ui.atk4.org/demos/basic/label.php

5.2.2.1 Basic Usage

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

```
$label = Label::addTo($app, ['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.

5.2.2.2 Icons

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

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

You can also specify icon as an object:

```
Label::addTo($app, ['new', 'iconRight' => new \Atk4\Ui\Icon('delete')]);
```

For more information, see: *Icon*

5.2.2.3 Image

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

```
$img = $app->cdn['atk'] . '/logo.png';
Label::addTo($app, ['Coded in PHP', 'image' => $img]);
```

5.2.2.4 Detail

You can specify "detail" component to your label:

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

5.2.2.5 Groups

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

```
$group = View::addTo($app, ['class.blue tag' => true, 'ui' => 'labels']);
Label::addTo($group, ['$9.99']);
Label::addTo($group, ['$19.99', 'class.red tag' => true]);
Label::addTo($group, ['$24.99']);
```

5.2.2.6 Combining classes

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

```
$columns = Columns::addTo($app);

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

// attach label to the top of left column
Label::addTo($col, ['Left Column', 'class.top attached' => true, 'icon' => 'book']);

// ribbon around left column
Label::addTo($col, ['Lorem', 'class.red ribbon' => true, 'icon' => 'cut']);

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

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

// attach label to the top of right column
Label::addTo($col, ['Right Column', 'class.top attached' => true, 'icon' => 'book']);

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

// right bottom corner label
Label::addTo($col, ['Ipsum', 'class.orange bottom right attached' => true, 'icon' => 'cut
↪']);
```

5.2.2.7 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->onHook(\Atk4\Ui\Table\Column::HOOK_GET_HTML_TAGS, function (Table $table, Model
↪$row) {
    if ($row->getId() == 1) {
        return [
            'name' => $table->getApp()->getTag('div', ['class' => 'ui ribbon label'],
↪$row->get('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.

5.2.3 Text

class Atk4\Ui**Text**

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

5.2.3.1 Basic Usage

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

```
$text = Text::addTo($app, ['here goes some text']);
```

5.2.3.2 Paragraphs

You can define multiple paragraphs with text like this:

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

5.2.3.3 HTML escaping

By default Text will escape HTML so this won't render as a bold text:

```
$text = Text::addTo($app, ['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 too:

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

5.2.3.4 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 paragraphs of text:

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

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

5.2.3.5 Limitations

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

5.2.4 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.

5.2.4.1 Basic Usage

```
LoremIpsum::addTo($app);
```

5.2.4.2 Resizing

You can define the length of the LoremIpsum text:

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

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

```
LoremIpsum::addTo($app, [1]); // just add a little one

// or

LoremIpsum::addTo($app, [5]); // adds a lot of text
```

5.2.5 Header

class Atk4\Ui**Header**

Can be used for page or section headers.

Based around: https://fomantic-ui.com/elements/header.html.

Demo: https://ui.atk4.org/demos/basic/header.php

5.2.5.1 Basic Usage

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

```
Header::addTo($this, ['Hello, Header']);
```

5.2.5.2 Attributes

property Atk4\Ui\Header::$**size**

property Atk4\Ui\Header::$**subHeader**

Specify size and sub-header content:

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

// or

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

5.2.5.3 Icon and Image

property Atk4\Ui\Header::$**icon**

property Atk4\Ui\Header::$**image**

Header may specify icon or image:

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

Here you can also specify seed for the image:

```
$img = $app->cdn['atk'] . '/logo.png';
Header::addTo($seg, [
    'Center-aligned header',
    'aligned' => 'center',
    'image' => [$img, 'class.disabled' => true],
    'subHeader' => 'header with image',
]);
```

5.2.6 Breadcrumb

class Atk4\Ui**Breadcrumb**

Implement navigational Breadcrumb, by using https://fomantic-ui.com/collections/breadcrumb.html

5.2.6.1 Basic Usage

Atk4\Ui\Breadcrumb::**addCrumb**()

Atk4\Ui\Breadcrumb::**set**()

Here is a simple usage:

```
$crumb = Breadcrumb::addTo($app);
$crumb->addCrumb('User', ['user']);
$crumb->addCrumb('Preferences', ['user_preferences']);
$crumb->set('Change Password');
```

Every time you call addCrumb a new one is added. With set() you can specify the name of the current page. addCrumb also requires a URL passed as second argument which can be either a string or array (which would then be passed to url() (`View::url`).

5.2.6.2 Changing Divider

property Atk4\Ui\Breadcrumb::$**dividerClass**

By default value `right angle icon` is used, but you can change it to `right chevron icon` or simply set to empty string "" to use slash.

5.2.6.3 Working with Path

property Atk4\Ui\Breadcrumb::$**path**

Atk4\Ui\Breadcrumb::**popTitle**()

Calling addCrumb adds more elements into the $path property. Each element there would contain 3 hash values:

- section - name that will appear to the user
- link - where to go if clicked
- divider - which divider to use after the crumb

By default `divider` is set to *Breadcrumb::$dividerClass*. You may also manipulate $path array yourself. For example the next code will use some logic:

```
$crumb = Breadcrumb::addTo($app);
$crumb->addCrumb('Users', []);

$model = new User($app->db);

$id = $app->stickyGet('user_id');
if ($id) {
    // perhaps we edit individual user?
```

(continues on next page)

(continued from previous page)

```
    $entity = $model->load($id);
    $crumb->addCrumb($entity->get('name'), []);

    // here we can check for additional criteria and display a deeper level on the crumb

    Form::addTo($app)->setEntity($entity);
} else {
    // display list of users
    $table = Table::addTo($app);
    $table->setModel($model);
    $table->addDecorator(['name', [\Atk4\Ui\Table\Column\Link::class, [], ['user_id' =>
↪'id']]);
}

$crumb->popTitle();
```

5.2.7 Icon

class Atk4\Ui**Icon**

Implements basic icon:

```
$icon = Icon::addTo($app, ['book']);
```

Alternatively:

```
$icon = Icon::addTo($app, [], ['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 Fomantic-UI icon documentation:

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

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

```
Button::addTo($app, ['Click Me', 'class.red' => true, 'icon' => 'flipped big question']);

Label::addTo($app, ['Battery Low', 'class.green' => true, 'icon' => 'battery low']);
```

5.2.7.1 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:

```
Header::addTo($app, ['Header', 'class.red' => true, 'icon' => 'flipped question']);
Button::addTo($app, ['Button', 'class.red' => true, 'icon' => 'flipped question']);

$menu = Menu::addTo($app);
$menu->addItem(['Menu Item', 'icon' => 'flipped question']);
$subMenu = $menu->addMenu(['Sub-menu', 'icon' => 'flipped question']);
$subMenu->addItem(['Sub Item', 'icon' => 'flipped question']);

Label::addTo($app, ['Label', 'class.right ribbon red' => true, 'icon' => 'flipped␣
↪question']);
```

5.2.7.2 Groups

Fomantic-UI support icon groups. The best way to implement is to supply HtmlTemplate to an icon:

```
Icon::addTo($app, ['template' => new \Atk4\Ui\HtmlTemplate('<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:

```
$view = View::addTo($app, ['template' => new \Atk4\Ui\HtmlTemplate('<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:

```
$icon = Icon::addTo($view, ['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:

```
$noUsers = new \Atk4\Ui\View(['class.huge icons' => true, 'element' => 'i']);
Icon::addTo($noUsers, ['big red dont']);
Icon::addTo($noUsers, ['black user']);

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

5.2.7.3 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\ViewWithContent
{
    public $social;
    public $icon;
    public $defaultTemplate; // __DIR__ . '../templates/socialadd.html'

    protected function init(): void
    {
        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;
        }
```

(continues on next page)

(continued from previous page)

```
        if (!$this->template) {
            // TODO: place template into file and set defaultTemplate instead
            $this->template = new \Atk4\Ui\HtmlTemplate(
'<{_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');
    }
}

// usage examples

// start with the most basic usage
SocialAdd::addTo($app, ['instagram']);

// next specify label and separately name of social network
SocialAdd::addTo($app, ['Follow on Twitter', 'social' => 'twitter']);

// finally provide custom icon and make the button clickable
SocialAdd::addTo($app, ['facebook', 'icon' => 'facebook f'])
    ->link('https://facebook.com', '_blank');
```

5.2.8 Image

class Atk4\Ui**Image**

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

5.2.8.1 Basic Usage

Implements basic image:

```
$icon = Image::addTo($app, ['image.gif']);
```

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

5.2.8.2 Specify classes

You can pass additional classes to an image:

```
$img = $app->cdn['atk'] . '/logo.png';
$icon = Image::addTo($app, [$img, 'class.disabled' => true]);
```

5.2.9 Message

class Atk4\Ui**Message**

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

Demo: https://ui.atk4.org/demos/basic/message.php

5.2.9.1 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', 'type' => 'warning']);
$app->add($message);
```

Here is the alternative syntax:

```
$message = Message::addTo($app, ['Warning Message Title', 'type' => 'warning']);
```

5.2.9.2 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:

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

5.2.9.3 Message Icon

property Atk4\Ui\Message::$**icon**

You can specify icon also:

```
$message = Message::addTo($app, [
    'Battery low',
    'class.red' => true,
    'icon' => 'battery low',
])->text->addParagraph('Your battery is getting low. Re-charge your Web App');
```

class Atk4\Ui**Tabs**

5.2.10 Tabs

Tabs implement a yet another way to organize your data. The implementation is based on: https://fomantic-ui.com/elements/icon.html.

Demo: https://ui.atk4.org/demos/interactive/tabs.php

5.2.10.1 Basic Usage

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

```
$tabs = Tabs::addTo($app);
```

Adding a static content is pretty simple:

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

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

Atk4\Ui\Tabs::**addTab**(*$name*, *$action = null*)

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 Closure action as a second parameter.

Example:

```
$tabs = Tabs::addTo($layout);

// add static tab
HelloWorld::addTo($tabs->addTab('Static Tab'));

// add dynamic tab
$tabs->addTab('Dynamically Loading', function (VirtualPage $vp) {
    LoremIpsum::addTo($vp);
});
```

5.2.10.2 Dynamic Tabs

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

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

```
$tabs = Tabs::addTo($app);

// dynamic tab
$tabs->addTab('Dynamic Lorem Ipsum', function (VirtualPage $vp) {
    LoremIpsum::addTo($vp, ['size' => 2]);
});

// dynamic tab
$tabs->addTab('Dynamic Form', function (VirtualPage $vp) {
    $mRegister = new \Atk4\Data\Model(new \Atk4\Data\Persistence\Array_($a));
    $mRegister->addField('name', ['caption' => 'Please enter your name (John)']);

    $form = Form::addTo($vp, ['class.segment' => true]);
    $form->setEntity($mRegister);
    $form->onSubmit(function (Form $form) {
        if ($form->entity->get('name') !== 'John') {
            return $form->jsError('name', 'Your name is not John! It is "' . $form->
↪entity->get('name') . '". It should be John. Pleeease!');
        }
    });
});
```

5.2.10.3 URL Tabs

Atk4\Ui\Tabs::**addTabUrl**(*$name*, *$url*)

Tab can load external URL or a different page if you prefer that instead of VirtualPage. This works similar to iframe:

```
$tabs = Tabs::addTo($app);

$tabs->addTabUrl('Terms and Condition', 'terms.html');
```

class Atk4\Ui**Accordion**

5.2.11 Accordion

Accordion implement another way to organize your data. The implementation is based on: https://fomantic-ui.com/modules/accordion.html.

Demo: https://ui.atk4.org/demos/interactive/accordion.php

5.2.11.1 Basic Usage

Once you create an Accordion container you can then mix and match static and dynamic accodion section:

```
$acc = Accordion::addTo($app);
```

Adding a static content section is pretty simple:

```
LoremIpsum::addTo($acc->addSection('Static Tab'));
```

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

Atk4\Ui\Accordion::**addSection**(*$name*, *$action* = *null*, *$icon* = *'dropdown'*)

Use addSection() method to add more section in an Accordion view. First parameter is a title of the section.

Section can be static or dynamic. Dynamic sections use *VirtualPage* implementation mentioned above. You should pass Closure action as a second parameter.

Example:

```
$t = Accordion::addTo($layout);

// add static section
HelloWorld::addTo($t->addSection('Static Content'));

// add dynamic section
$t->addSection('Dynamically Loading', function (VirtualPage $vp) {
    LoremIpsum::addTo($vp);
});
```

5.2.11.2 Dynamic Accordion Section

Dynamic sections are based around implementation of *VirtualPage* and allow you to pass a callback which will be triggered when user clicks on the section title.:

```
$acc = Accordion::addTo($app);

// dynamic section
$acc->addSection('Dynamic Lorem Ipsum', function (VirtualPage $vp) {
    LoremIpsum::addTo($vp, ['size' => 2]);
});
```

5.2.11.3 Controlling Accordion Section via Javascript

Accordion class has some wrapper method in order to control the accordion module behavior.

Atk4\Ui\Accordion::**jsOpen**(*$section*, *$action* = *null*)

Atk4\Ui\Accordion::**jsToggle**(*$section*, *$action* = *null*)

Atk4\Ui\Accordion::**jsClose**(*$section*, *$action* = *null*)

For example, you can set a button that, when clicked, will toggle an accordion section:

```
$button = Button::addTo($bar, ['Toggle Section 1']);

$acc = Accordion::addTo($app, ['type' => ['styled', 'fluid']]);
$section1 = LoremIpsum::addTo($acc->addSection('Static Text'));
$section2 = LoremIpsum::addTo($acc->addSection('Static Text'));

$button->on('click', $acc->jsToggle($section1));
```

5.2.11.4 Accordion Module settings

It is possible to change Accordion module settings via the settings property.:

```
Accordion::addTo($app, ['settings' => []]);
```

For a complete list of all settings for the Accordion module, please visit: https://fomantic-ui.com/modules/accordion.html#/settings

5.2.12 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".

5.2.12.1 Basic Usage

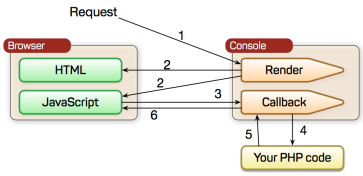
To add a "Hello, World" message:

```
HelloWorld::addTo($app);
```

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.

5.3 Interactive components

Interactive components rely on *Callbacks*, *VirtualPage* or *Server-Sent Events (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 interactive components will typically communicate:

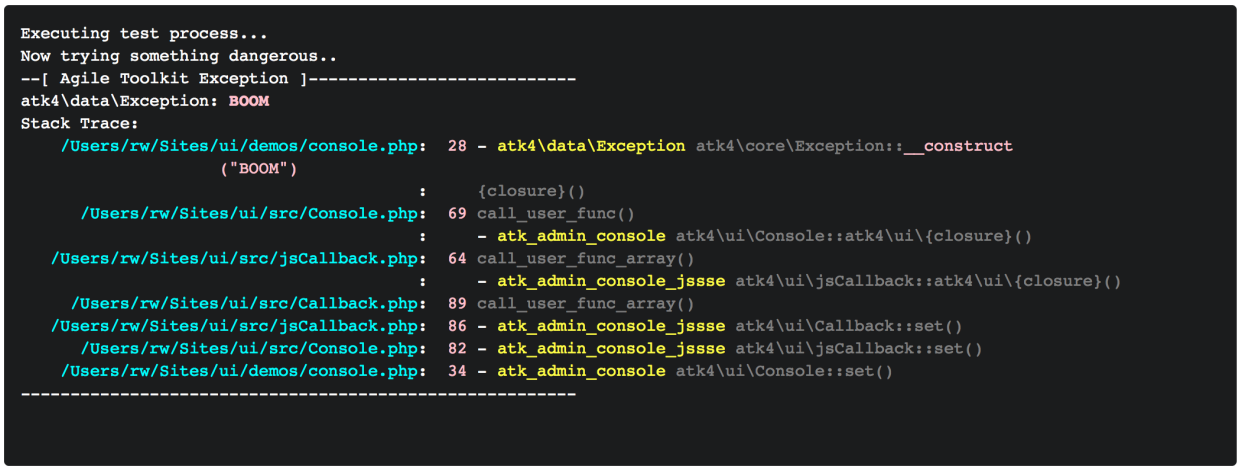


1. request by browser is made.
2. *App* asks *Console* to render HTML + JavaScript.
3. JavaScript invokes AJAX request using a *Callback* URL.
4. Callback invokes user-defined PHP code, which will generate some `Console::output()`.
5. Response is encoded and

6. sent back to the browser either as JSON or *Server-Sent Events (JsSse)*.

class Atk4\Ui**Console**

5.3.1 Console



With console you can output real-time information to the user directly from PHP. It can be used do direct output from slow method or even execute commands on the server (such as `ping`).

Demo: https://ui.atk4.org/demos/interactive/console.php

5.3.1.1 Basic Usage

Atk4\Ui\Console::**set**(*$callback*)

Atk4\Ui\Console::**send**(*$callback*)

After adding a console to your *Render Tree*, you just need to set a callback:

```
$console = Console::addTo($app);
$console->set(function (Console $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\Js\JsExpression('alert([])', ['The wait is over']));
});
```

Console uses *Server-Sent Events (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. JavaScript in a browser captures real-time events and displays it on a black background.

Console integrates nicely with DebugTrait (https://atk4-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.

5.3.1.2 Using With Object

`Atk4\Ui\Console::`**`runMethod`**`(`*`$callback`*`)`

We recommend that you pack up your busineess 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:

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

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

```
use \Atk4\Core\DebugTrait();

public function generateReport($pages)
{
    $this->info('converting report to PDF');

    // slow stuff
    $this->info('almost done, be patient..');

    // more slow stuff
    return true;
}
```

You can also execute static methods:

```
$console->runMethod('StaticLib', 'myStaticMethod');
```

5.3.1.3 Executing Commands

`Atk4\Ui\Console::`**`exec`**`(`*`$cmd`*`, `*`$args`*`)`

`property` `Atk4\Ui\Console::$`**`lastExitCode`**

To execute a command, use:

```
$console->exec('/sbin/ping', ['-c', '5', '-i', '1', '192.168.0.1']);
```

This will run a command, and will stream command output to you. Console is implemented to capture both STDOUT and STDERR in real-time then display it on the console using color. Console does not support ANSI output.

Method exec can be executed directly on the $console or inside the callback:

```
$console->set(function (Console $console) {
    $console->eval();
});
```

Without callback, eval will wrap itself into a callback but you can only execute a single command. When using callback form, you can execute multiple commands:

```
Console::addTo($app)->set(function (Console $c) {
    $c
        ->exec('/sbin/ping', ['-c', '5', '-i', '1', '192.168.0.1'])
        ->exec('/sbin/ping', ['-c', '5', '-i', '2', '8.8.8.8'])
        ->exec('/bin/no-such-command');
});
```

Method exec() will return `$this` if command was run inside callback and was successful. It will return `false` on error and will return `null` if called outside of callback. You may also refer to *Console::$lastExitCode* which contains exit code of the last command.

Normally it's safe to chain `exec` which ensures that execution will stack. Should any command fail, the subsequent `exec` won't be performed.

NOTE that for each invocation `exec` will spawn a new process, but if you want to execute multiple processes, you can wrap them into `bash -c`:

```
Console::addTo($app)->exec('bash', [
    '-c',
    'cd ..; echo \'Running "composer update" in `pwd`\'; composer --no-ansi update; echo␣
↪\'Self-updated. OK to refresh now!\'',
]);
```

This also demonstrates argument escaping.

class Atk4\Ui**ProgressBar**

5.3.2 ProgressBar

ProgressBar is actually a quite simple element, but it can be made quite interactive along with *JsSse*.

Demo: https://ui.atk4.org/demos/interactive/progress.php

5.3.2.1 Basic Usage

Atk4\Ui\ProgressBar::**jsValue**(*$value*)

After adding a console to your *Render Tree*, you just need to set a callback:

```
// add progressbar showing 0 (out of 100)
$bar = ProgressBar::addTo($app);

// add with some other value of 20% and label
$bar2 = ProgressBar::addTo($app, [20, '% Battery']);
```

The value of the progress bar can be changed either before rendering, inside PHP, or after rendering with JavaScript:

```
$bar->value = 5; // sets this value instead of 0

Button::addTo($app, ['charge up the battery'])
    ->on('click', $bar2->jsValue(100));
```

5.3.2.2 Updating Progress in RealTime

You can use real-time element such as JsSse or Console (which relies on JsSse) to execute jsValue() of your progress bar and adjust the display value.

Demo: https://ui.atk4.org/demos/interactive/sse.php

Console also implements method *Console::send* so you can use it to send progress updates of your progress-bar.

5.3.3 Popup

class Atk4\Ui**Popup**

Implements a popup:

```
$button = Button::addTo($app, ['Click me']);
HelloWorld::addTo(Popup::addTo($app, [$button]));
```

Atk4\Ui\Popup::**set**(*$callback*)

Popup can also operate with dynamic content:

```
$button = Button::addTo($app, ['Click me']);
Popup::addTo($app, [$button])
    ->set('hello world with rand=' . rand(1, 100));
```

Pop-up should be added into a viewport which will define boundaries of a pop-up, but it will be positioned relative to the $button. Popup remains invisible until it's triggered by event of $button.

If second argument in the *Seed* is of class *Button*, Menu, MenuItem or Dropdown (note - NOT Form\Control!), pop-up will also bind itself to that element. The above example will automatically bind "click" event of a button to open a pop-up.

When added into a menu, pop-up will appear on hover:

```
$menu = Menu::addTo($app);
$item = $menu->addItem('HoverMe')
Text::addTo(Popup::addTo($app, [$item]))->set('Appears when you hover a menu item');
```

Like many other Views of ATK, popup is an interactive element. It can load it's contents when opened:

```
$menu = Menu::addTo($app);
$item = $menu->addItem('HoverMe');
Popup::addTo($app, [$item])->set(function (View $p) {
    Text::addTo($p)->set('Appears when you hover a menu item');
    Label::addTo($p, ['Random value', 'detail' => rand(1, 100)]);
});
```

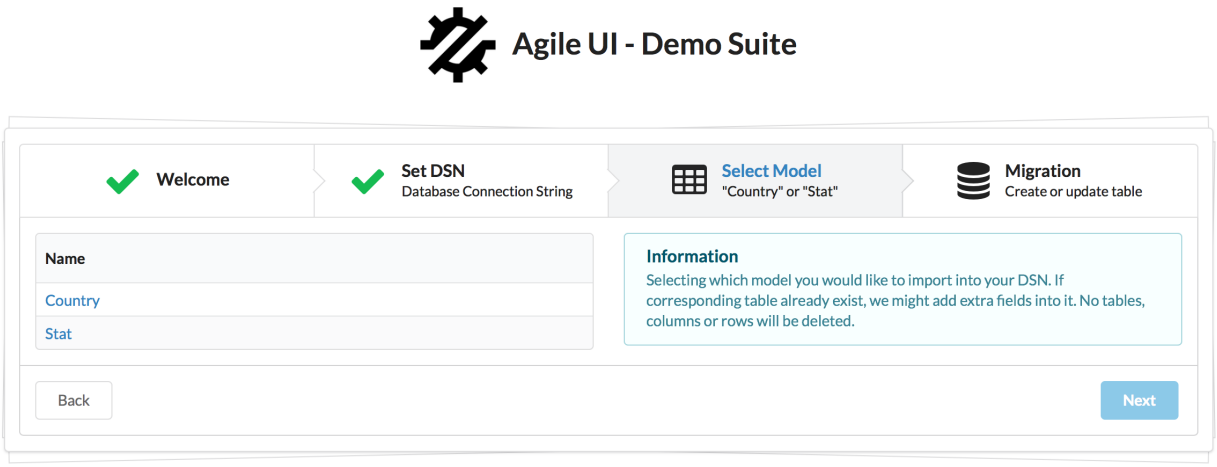
Demo: https://ui.atk4.org/demos/interactive/popup.php

Fomantic-UI: https://fomantic-ui.com/modules/popup.html

class Atk4\Ui**Wizard**

5.3.4 Wizard

Wizard is a high-level component, which makes use of callback to track step progression through the stages. It has an incredibly simple syntax for building UI and display a lovely UI for you.



Demo: https://ui.atk4.org/demos/interactive/wizard.php

Introduced in UI v1.4

5.3.4.1 Basic Usage

`Atk4\Ui\Wizard::`**`addStep`**(*$title*, *$callback*)

`Atk4\Ui\Wizard::`**`addFinish`**(*$callback*)

Start by creating Wizard inside your render tree:

```
$wizard = Wizard::addTo($app);
```

Next add as many steps as you need specifying title and a PHP callback code for each:

```
$wizard->addStep('Welcome', function (Wizard $wizard) {
    Message::addTo($wizard, ['Welcome to wizard demonstration'])->text
        ->addParagraph('Use button "Next" to advance')
        ->addParagraph('You can specify your existing database connection string which␣
↪will be used'
            . ' to create a table for model of your choice');
});
```

Your callback will also receive `$wizard` as the first argument. Method addStep returns *WizardStep*, which is described below. You can also provide first argument to addStep as a seed or an object:

```
$wizard->addStep([
    'Set DSN',
    'icon' => 'configure',
    'description' => 'Database Connection String',
], function (Wizard $p) {
```

(continues on next page)

(continued from previous page)

```
    // some code here
});
```

You may also specify a single Finish callback, which will be used when wizard is complete:

```
$wizard->addFinish(function (Wizard $wizard) {
    Header::addTo($wizard, ['You are DONE', 'class.huge centered' => true]);
});
```

5.3.4.2 Properties

When you create wizard you may specify some of the following options:

property Atk4\Ui\Wizard::$**defaultIcon**

Other properties are used during the execution of the wizard.

5.3.4.3 Step Tracking

property Atk4\Ui\Wizard::$**stepCallback**

Wizard employs *Callback* to maintain which step you currently are on. All steps are numbered started with 0.

Important: Wizard currently does not enforce step completion. Changing step number in the URL manually can take you to any step. You can also go backwards and re-do steps. Section below explains how to make wizard enforce some restrictions.

---

property Atk4\Ui\Wizard::$**currentStep**

When Wizard is initialized, it will set currentStep to a number (0, 1, 2, ..) corresponding to your steps and finish callback, if you have specified it.

property Atk4\Ui\Wizard::$**buttonPrevious**

property Atk4\Ui\Wizard::$**buttonNext**

property Atk4\Ui\Wizard::$**buttonFinish**

Those properties will be initialized with the buttons, but some of them may be destroyed by the render step, if the button is not applicable. For example, first step should not have "Previous" button. You can change label or icon on existing button.

5.3.4.4 Code Placement

As you build up your wizard, you can place code inside callback or outside. It will have a different effect on your wizard:

```
$wizard->buttonNext->icon = 'person';

$wizard->addStep('Step 3', function (Wizard $wizard) {
    $wizard->buttonNext->icon = 'book';
});
```

Step defines the callback and will execute it instantly if the step is active. If step 3 is active, the code is executed to change icon to the book. Otherwise icon will remain 'person'. Another handy technique is disabling the button by adding "disabled" class.

5.3.4.5 Navigation

Wizard has few methods to help you to navigate between steps.

`Atk4\Ui\Wizard::urlNext()`

`Atk4\Ui\Wizard::jsNext()`

Methods starting with `url` will return a URL towards the next step. jsNext() method returns javascript action which will take you to the next step.

If you wish to to go to specific step, you can use `$wizard->stepCallback->getUrl($step);`

Finally you can get URL of the current step with `$wizard->url()` (see `View::url`)

5.3.4.6 WizardStep

`class Atk4\Ui\WizardStep`

`property Atk4\Ui\WizardStep::$title`

`property Atk4\Ui\WizardStep::$description`

`property Atk4\Ui\WizardStep::$icon`

`property Atk4\Ui\WizardStep::$wizard`

Each step of your wizard serves two roles. First is to render title and icon above the wizard and second is to contain a callback code.

5.3.5 Right Panel

`class Atk4\Ui\Panel\Right`

Right panel are view attached to the app layout. They are display on demand via javascript event and can display content statically or dynamically using Loadable Content.

Demo: https://ui.atk4.org/demos/layout/layout-panel.php

5.3.5.1 Basic Usage

Adding a right panel to the app layout and adding content to it:

```
$panel = $app->layout->addRightPanel(new \Atk4\Ui\Panel\Right(['dynamic' => false]));
Message::addTo($panel, ['This panel contains only static content.']);
```

By default, panel content are loaded dynamically. If you want to only add static content, you need to specify the dynamic property and set it to false.

Opening of the panel is done via a javascript event. Here, we simply register a click event on a button that will open the panel:

```
$button = Button::addTo($app, ['Open Static']);
$button->on('click', $panel->jsOpen());
```

Loading content dynamically

Loading dynamic content within panel is done via the onOpen method

Atk4\Ui\Panel\Right::**onOpen**(*$callback*)

Initializing a panel with onOpen callback:

```
$panel = $app->layout->addRightPanel(new \Atk4\Ui\Panel\Right());
Message::addTo($panel, ['This panel will load content dynamically below according to␣
↪button select on the right.']);
$button = Button::addTo($app, ['Button 1']);
$button->js(true)->data('btn', '1');
$button->on('click', $panel->jsOpen(['btn'], 'orange'));

$panel->onOpen(function (Panel\Content $p) {
    $buttonNumber = $p->getApp()->tryGetRequestQueryParam('btn');
    $text = 'You loaded panel content using button #' . $buttonNumber;
    Message::addTo($p, ['Panel 1', 'text' => $text]);
});
```

Atk4\Ui\Panel\Right::**jsOpen**()

This method may take up to three arguments.

Parameters

- **$args** – an array of data property to carry with the callback URL. Let's say that you triggering element as a data property name ID (data-id) then if specify, the data ID value will be sent as a get argument with the callback URL.
- **$activeCss** – a string representing the active state of the triggering element. This CSS class will be applied to the trigger element as long as the panel remains open. This help visualize, which element has trigger the panel opening.
- **$jsTrigger** – JS expression that represent the jQuery object where the data property reside. Default to $(this).

5.3.6 Data Action Executor

Data action executor in UI is parts of interactive components that can execute a Data model defined user action. For more details on Data Model User Action please visit: https://atk4-data.readthedocs.io/en/develop/model.html#actions

Atk UI offers many types of action executor. A model user action may contain many properties. Usually, you would choose the type of executor based on the action definition. For example, an action that would required arguments prior to be executed can be set using an ArgumentFormExecutor. Or actions that can run using a single button can use a JsCallbackExecutor.

Demo: https://ui.atk4.org/demos/data-action/actions.php

5.3.6.1 Executor Interface

All executors must implement the ExecutorInterface or JsExecutorInterface interface.

interface Atk4\Ui**UserAction\ExecutorInterface**

interface Atk4\Ui**UserAction\JsExecutorInterface**

5.3.6.2 Basic Executor

class Atk4\Ui**UserAction\BasicExecutor**

This is the base view for most of the other action executors. This executor generally required that necessary arguments needed to run the action has been set. BasicExecutor will display:

- a button for executing the action;
- a header where action name and description are displayed;
- an error message if an action argument is missing;

5.3.6.3 Preview Executor

class Atk4\Ui**UserAction\PreviewExecutor**

This executor is specifically set in order to display the $preview property of the current model UserAction. You can select to display the preview using regular console type container, regular text or using HTML content.

5.3.6.4 Form Executor

class Atk4\Ui**UserAction\FormExecutor**

This executor will display a form where user is required to fill in either all model fields or certain model fields depending on the model UserAction $field property. Form control will depend on model field type.

5.3.6.5 Argument Form Executor

class Atk4\Ui**UserAction\ArgumentFormExecutor**

This executor will display a form but instead of filling form control with model field, it will use model UserAction $args property. This is used when you need to ask user about an argument value prior to execute the action. The type of form control type to be used in form will depend on how $args is setup within the model UserAction.

5.3.6.6 JS Callaback Executor

class Atk4\Ui**UserAction\JsCallbackExecutor**

This type of executor will output proper javascript that you can assign to a view event using View::on() method. It is also possible to pass the UserAction argument via $_POST argument.

5.3.6.7 Modal Executor

class Atk4\Ui**UserAction\ModalExecutor**

The ModalExecutor is base on Modal view. This is a one size fits all for model UserAction. When setting the UserAction via the ModelExecutor::setAction($action) method, it will automatically determine what step is require and will display each step base on the action definition within a modal view:

- Step 1: Argument definition. If the action required arguments, then the modal will display a form and ask user to fill argument values required by the model UserAction;
- Step 2: Field definition. If the action required fields, then the modal will display a form and ask user to fill field values required by the model UserAction;
- Step 3: Preview. If the action preview is set, then the modal will display it prior to execute the action.

The modal title default is set from the UserAction::getDescription() method but can be override using the Modal::$title property.

5.3.6.8 Confirmation Executor

class Atk4\Ui**UserAction\ConfirmationExecutor**

Like ModalExecutor, Confirmation executor is also based on a Modal view. It allow to display UserAction::confirmation property prior to execute the action. Since UserAction::confirmation property may be set with a Closure function, this give a chance to return specific record information to be displayed to user prior to execute the action.

Here is an example of an user action returning specific record information in the confirmation message:

```
$country->addUserAction('delete_country', [
    'caption' => 'Delete',
    'description' => 'Delete Country',
    'ui' => ['executor' => [\Atk4\Ui\UserAction\ConfirmationExecutor::class]],
    'confirmation' => function (Model\UserAction $action) {
        return 'Are you sure you want to delete this country: $action->getModel()->
↪getTitle();
    },
    'callback' => 'delete',
]);
```

The modal title default is set from the UserAction::getDescription() method but can be override using the Modal::$title property.

5.3.6.9 Executor HOOK_AFTER_EXECUTE

Executors can use the HOOK_AFTER_EXECUTE hook in order to return javascript action after the model UserAction finish executing. It is use in Crud for example in order to display users of successful model UserAction execution. Either by displaying Toast messages or removing a row within a Crud table.

Some Ui View component, like Crud for example, will also set javascript action to return based on the UserAction behavior. For example if the action deleted an entity then Crud will delete a table row. If the action updated an entity then Table needs to be reloaded.

5.3.6.10 The Executor Factory

class Atk4\Ui**UserAction\ExecutorFactory**

property Atk4\Ui\UserAction\ExecutorFactory::$**executorSeed**

Executor factory is responsible for creating proper executor type in regards to the model user action being used.

The factory createExecutor method:

```
ExecutorFactory::createExecutor(UserAction $action, View $owner, $requiredType = null)
```

Based on parameter passed to the method, it will return proper executor for the model user action.

If $requiredType is set, then it will look for basic type executor already register in $executorSeed property for that specific type.

When required is not set, it will first look for a specific executor that has been already register for the model/action.

If no executor type is found, then the createExecutor method will determine one, based on the model user action properties:

- if action contains a callable confirmation property, then, the executor create is based on CONFIRMATION_EXECUTOR type;
- if action contains use either, fields, argument or preview properties, then, the executor create is based on MODAL_EXECUTOR type;
- if action does not use any of the above properties, then, the executor create is based on JS_EXECUTOR type.

The createExecutor method also add the executor to the View passed as argument. However, note that when an executor View parent class is of type Modal, then it will be attached to the $app->html view instead. This is because Modal view in ui needs to be added to $app->html view in order to work correctly on reload.

Changing or adding Executor type

Existing executor type can be change or added globally for all your user model actions via this method:

```
ExecutorFactory::registerTypeExecutor(string $type, array $seed): void
```

This will set a type to your own executor class. For example, a custom executor class can be set as a MODAL_EXECUTOR type and all model user action that use this type will be executed using this custom executor instance.

Type may also be registered per specific model user action via this method:

```
ExecutorFactory::registerExecutor(UserAction $action, array $seed): void
```

For example, you need a custom executor to be created when using a specific model user action:

```
class MySpecialFormExecutor extends \Atk4\Ui\UserAction\ModalExecutor
{
    public function addFormTo(\Atk4\Ui\View $view): \Atk4\Ui\Form
    {
        $myView = MySpecialView::addTo($view);

        return parent::addFormTo($myView);
    }
```

(continues on next page)

(continued from previous page)

```
}

// ...
ExecutorFactory::registerExecutor($action, [MySpecialFormExecutor::class]);
```

Then, when ExecutorFactory::createExecutor method is called for this $action, MySpecialExecutor instance will be create in order to run this user model action.

Triggering model user action

The Executor factory is also responsible for creating the UI view element, like regular, table or card button or menu item that will fire the model user action execution.

The method is:

```
ExecutorFactory::createTrigger(UserAction $action, string $type = null): View
```

This method return an instance object for the proper type. When no type is supply, a default View Button object is returned.

As per executor type, it is also possible to add or change already register type via the registerTrigger method:

```
ExecutorFactory::registerTrigger(string $type, $seed, UserAction $action, bool
↪$isSpecific = false): void
```

Again, the type can be apply globally to all action using the same name or specifically for a certain model/action.

For example, changing default Table button for a specific model user action when this action is used inside a crud table:

```
ExecutorFactory::registerTrigger(
    ExecutorFactory::TABLE_BUTTON,
    [Button::class, null, 'icon' => 'mail'],
    $m->getUserAction('mail')
);
```

This button view will then be display in Crud when it use a model containing 'mail' user action.

Overriding ExecutorFactory

Overriding the ExecutorFactory class is a good way of changing the look of all trigger element within your app or within a specific view instance.

Example of changing button for Card, Crud and Modal executor globally within your app:

```
class MyFactory extends \Atk4\Ui\UserAction\ExecutorFactory
{
    protected static $actionTriggerSeed = [
        self::MODAL_BUTTON => [
            'edit' => [Button::class, 'Save', 'class.green' => true],
            'add' => [Button::class, 'Save', 'class.green' => true],
        ],
        self::TABLE_BUTTON => [
            'edit' => [Button::class, null, 'icon' => 'pencil'],
```

(continues on next page)

(continued from previous page)

```
            'delete' => [Button::class, null, 'icon' => 'times red'],
        ],
        self::CARD_BUTTON => [
            'edit' => [Button::class, 'Edit', 'icon' => 'pencil', 'ui' => 'tiny button'],
            'delete' => [Button::class, 'Remove', 'icon' => 'times', 'ui' => 'tiny button
↪'],
        ],
    ];

    protected static $actionCaption = [
        'add' => 'Add New Record',
    ];
}

// ...
$app->defaultExecutorFactory = $myFactory;
```

5.3.6.11 Model UserAction assignment to View

It is possible to assign a model UserAction to the View::on() method directly:

```
$button->on('click', $model->getUserAction('my_action'));
```

By doing so, the View::on() method will automatically determine which executor is required to properly run the action. If the model UserAction contains has either $fields, $args or $preview property set, then the ModalExecutor will be used, JsCallback will be used otherwise.

It is possible to override this behavior by setting the $ui['executor'] property of the model UserAction, since View::on() method will first look for that property prior to determine which executor to use.

Example of overriding executor assign to a button.:

```
$myAction = $model->getUserAction('my_action');
$myAction->ui['executor'] = $myExecutor;

$button->on('click', $myAction);
```

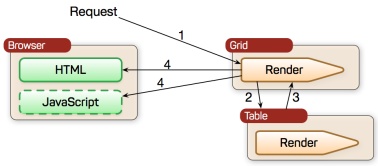
Demo

For more information on how Model UserAction are assign to button and interact with user according to their definition, please visit: Assign action to button event

You will find the UserAction definition for the demo here

5.4 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()*.

5.4.1 Crud

class Atk4\Ui**Crud**

Crud class offers a very usable extension to *Grid* class, which automatically adds actions for deleting, updating and adding records as well as linking them with corresponding Model actions.

Important: If you only wish to display a non-interactive table use *Table* class. If you need to display Data Grid with some custom actions (not update/delete/add) or if you want to use your own editing mechanism (such as edit data on separate page, not inside a modal), use *Grid*

Important: ATK Addon - MasterCrud implements a higher-level multi-model management solution, that takes advantage of model relations and traversal to create multiple levels of Cruds: https://github.com/atk4/mastercrud

---

5.4.1.1 Using Crud

The basic usage of Crud is:

```
Crud::addTo($app)->setModel(new Country($app->db));
```

Users are now able to fully interact with the table. There are ways to restrict which "rows" and which "columns" user can access. First we can only allow user to read, manage and delete only countries that are part of European Union:

```
$euCountries = new Country($app->db);
$euCountries->addCondition('is_eu', true);

Crud::addTo($app)->setModel($euCountries);
```

After that column `is_eu` will not be editable to the user anymore as it will be marked `system` by `addCondition`.

You can also specify which columns you would like to see on the grid:

```
$crud->setModel($euCountries, ['name']);
```

This restriction will apply to both viewing and editing, but you can fine-tune that by specifying one of many parameters to Crud.

5.4.1.2 Disabling Actions

By default Crud allows all four operations - creating, reading, updating and deleting. These action is set by default in model action. It is possible to disable these default actions by setting their system property to true in your model:

```
$euCountries->getUserAction('edit')->system = true;
```

Model action using system property set to true, will not be display in Crud. Note that action must be setup prior to use `$crud->setModel($euCountries)`

5.4.1.3 Specifying Fields (for different views)

property Atk4\Ui\Crud::$**displayFields**

Only fields name set in this property will be display in Grid. Leave empty for all fields.

property Atk4\Ui\Crud::$**editFields**

If you'd like to have different fields in the grid of the CRUD, but you need more/different fields in the editing modal (which opens when clicking on an entry), you can choose here the fields that are available in the editing modal window.

Important: Both views (overview and editing view) refer to the same model, just the fields shown in either of them differ

---

Example:

```
$crud = \Atk4\Ui\Crud::addTo($app);
$model = new \Atk4\Data\Model($app->db);
$crud->displayFields(['field1, field2']);
$crud->editFields(['field1, field2, field3, field4']);
```

property Atk4\Ui\Crud::$**addFields**

Through those properties you can specify which fields to use when form is display for add and edit action. Field name add here will have priorities over the action fields properties. When set to null, the action fields property will be used.

5.4.1.4 Custom Form Behavior

Form in Agile UI allows you to use many different things, such as custom layouts. With Crud you can specify your own form behavior using a callback for action:

```
// callback for model action add form
$g->onFormAdd(function (Form $form, ModalExecutor $ex) {
    $form->js(true, $form->getControl('name')->jsInput()->val('Entering value via␣
↪javascript'));
});

// callback for model action edit form
$g->onFormEdit(function (Form $form, ModalExecutor $ex) {
    $form->js(true, $form->getControl('name')->jsInput()->attr('readonly', true));
});

// callback for both model action edit and add
```

(continues on next page)

(continued from previous page)

```
$g->onFormAddEdit(function (Form $form, ModalExecutor $ex) {
    $form->onSubmit(function (Form $form) use ($ex) {
        return new \Atk4\Ui\Js\JsBlock([
            $ex->jsHide(),
            new \Atk4\Ui\Js\JsToast('Submit all right! This demo does not saved data.'),
        ]);
    });
});
```

Callback function will receive the Form and ActionExecutor as arguments.

5.4.1.5 Changing titles

Important: Changing the title of the CRUD's grid view must be done before setting the model. Changing the title of the modal of a CRUD's modal window must be done after loading the model. Otherwise the changes will have no effect.

Here's an example:

```
$crud = \Atk4\Ui\Crud::addTo($app);
$model = new \Atk4\Data\Model($app->db);
$model->getUserAction('add')->description = 'New title for adding a record'; // the␣
↪button of the overview - must be loaded before setting model
$crud->setModel($model);
$model->getUserAction('add')->ui['executor']->title = 'New title for modal'; // the␣
↪button of the modal - must be rendered after setting model
```

5.4.1.6 Notification

property Atk4\Ui\Crud::**$notifyDefault**

property Atk4\Ui\Crud::**$saveMsg**

property Atk4\Ui\Crud::**$deleteMsg**

property Atk4\Ui\Crud::**$defaultMsg**

When a model action execute in Crud, a notification to user is display. You can specify your notifier default seed using `$notifyDefault`. The notifier message may be set via `$saveMsg`, `$deleteMsg` or `$defaultMsg` property.

5.4.2 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.

5.4.2.1 Using Grid

Here is a simple usage:

```
Grid::addTo($app)->setModel(new Country($db));
```

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

```
$grid = Grid::addTo($app);
$grid->setModel(new Country($db));

$grid->addQuickSearch();
$grid->menu->addItem('Reload Grid', new \Atk4\Ui\Js\JsReload($grid));
```

5.4.2.2 Adding Menu Items

property Atk4\Ui\Grid::$**menu**

Atk4\Ui\Grid::**addButton**(*$label*)

Grid top-bar which contains QuickSearch is implemented using Fomantic-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 Fomantic-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:

```
$grid = Grid::addTo($app, ['menu' => false]);
```

5.4.2.3 Adding Quick Search

property Atk4\Ui\Grid::$**quickSearch**

Atk4\Ui\Grid::**addQuickSearch**(*$fields = []*, *$hasAutoQuery = false*)

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. (https://atk4-data.readthedocs.io/en/develop/model.html#title-field)

By default, quick search input field will query server when user press the Enter key. However, it is possible to make it querying the server automatically, i.e. after the user has finished typing, by setting the auto query parameter:

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

5.4.2.4 Paginator

property Atk4\Ui\Grid::$**paginator**

property Atk4\Ui\Grid::$**ipp**

Grid comes with a paginator already. You can disable it by setting $paginator property to false. Alternatively you can provide seed for the paginator or even entire object:

```
$grid = Grid::addTo($app, ['paginator' => ['range' => 2]]);
```

You can use $ipp property to specify different number of items per page:

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

JsPaginator

Atk4\Ui\Grid::**addJsPaginator**(*$ipp*, *$options = []*, *$container = null*, *$scrollRegion = 'Body'*)

JsPaginator will load table content dynamically when user scroll down the table window on screen.

```
$table->addJsPaginator(30);
```

See *Table::addJsPaginator*

Atk4\Ui\Grid::**addJsPaginatorInContainer**(*$ipp*, *$containerHeight*, *$options = []*, *$container = null*, *$scrollRegion = 'Body'*)

Use this method if you want fixed table header when scrolling down table. In this case you have to set fixed height of your table container.

5.4.2.5 Actions

property Atk4\Ui\Grid::$**actions**

Atk4\Ui\Grid::**addActionButton**(*$button*, *$action*, *$confirm = false*)

Table supports use of Table\Column\Actions, which allows to display button for each row. Calling addActionButton() 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 Table\Column\Actions::addAction

Atk4\Ui\Grid::**addModalAction**(*$button*, *$title*, *$callback*)

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 (View $p, $id) use (
↪$grid) {
    // $id of the record which was clicked
    // $entity = $grid->model->load($id);
```

(continues on next page)

(continued from previous page)

```
    LoremIpsum::addTo($p);
});
```

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

See `Table\Column\Actions::addModal`

5.4.2.6 Column Menus

`Atk4\Ui\Grid::`**addDropdown**(*$columnName*, *$items*, *$fx*, *$icon = 'caret square down'*, *$menuId = null*)

`Atk4\Ui\Grid::`**addPopup**(*$columnName*, *$popup = null*, *$icon = 'caret square down'*)

Methods addDropdown and addPopup provide a wrapper for *Table\Column::addDropdown* and *Table\Column::addPopup* methods.

5.4.2.7 Selection

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

```
$sel = $grid->addSelection();
$grid->menu->addItem('show selection')
    ->on('click', new \Atk4\Ui\Js\JsExpression(
        'alert(\'Selected: \' + [])', [$sel->jsChecked()]
    ));
```

5.4.2.8 Sorting

property `Atk4\Ui\Grid::$`**sortable**

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

You can also set $sortable property for each table column decorator. That way you can enable/disable sorting of particular columns.

See also *Table::$sortable*.

5.4.2.9 Advanced Usage

property `Atk4\Ui\Grid::$`**table**

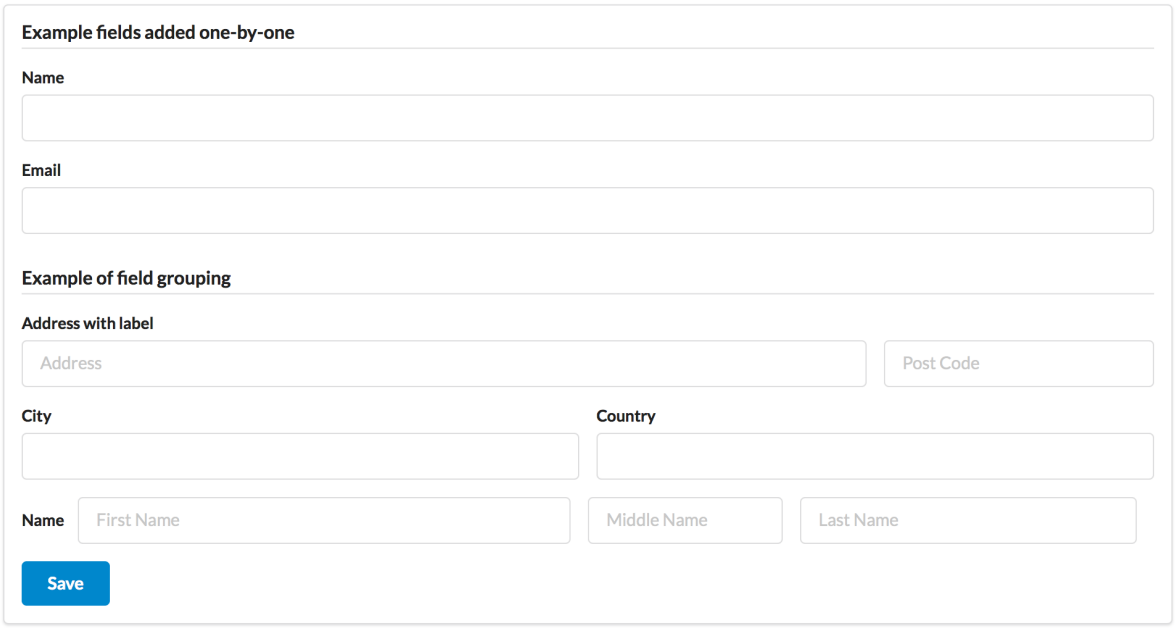
You can use a different component instead of default *Table* by injecting $table property.

5.4.3 Forms

class Atk4\Ui**Form**

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

- Form Rendering using Fomantic-UI HTML/CSS (https://fomantic-ui.com/collections/form.html):



- Form controls are automatically populated based on your existing data model with special treatment for date/time, auto-complete and even file upload.
- Loading data from database and storing it back. Any persistence (SQL, NoSQL) supported by ATK Data (https://atk4-data.readthedocs.io/en/develop/persistence.html) can be used.
- Support for Events and Actions on form controls, buttons and form callback. (*JavaScript Mapping*) Automatic execution of PHP-based Submit Handler passing all the collected data (*Callbacks*)

So if looking for a PHP Form class, ATK Form has the most complete implementation which does not require to fall-back into HTML / JS, perform any data conversion, load / store data and implement any advanced interactions such as file uploads.

5.4.3.1 Basic Usage

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

```
$form = Form::addTo($app);
$form->addControl('email');
```

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

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

(continues on next page)

(continued from previous page)

```
    return "Subscribed " . $form->entity->get('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';
```

or you can tweak it when you create form like this:

```
$form = Form::addTo($app, ['buttonSave' => [null, 'Subscribe', 'icon' => 'mail']]);
```

To set the default values in the form controls you can use the model property of the form. Even if model not explicitly set (see section below) each form has an underlying model which is automatically generated:

```
// single field
$form->entity->set('email', 'some@email.com');

// or multiple fields
$form->entity->set([
    'name' => 'John',
    'email' => 'some@email.com',
]);
```

Form also relies on a `\Atk4\Ui\Form::Layout` class and displays form controls through decorators defined at `\Atk4\Ui\Form::Control`. See dedicated documentation for:

- *Form\Layout*
- `Form\Control`

To tweak the UI properties of an form control input use `setInputAttr()` (and not the surrounding `<div>` as `setAttr()` would do). Here is how to set the HTML "maxlength" attribute on the generated input field:

```
$form = \Atk4\Ui\Form::addTo($this);
$form->setEntity($model);
$form->getControl('name')->setInputAttr('maxlength', 20);
```

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 (https://atk4-data.readthedocs.io/en/develop/model.html):

```
// Form will automatically add a new user and save into the database
$form = Form::addTo($app);
$form->setEntity(new User($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 the field type, a form control will be selected from Form\Control namespace
- Using `Form\Layout\Columns` can make form more compact by splitting it into columns
- Form control captions, placeholders, hints and other elements defined in Field::ui are respected (https://atk4-data.readthedocs.io/en/develop/fields.html#Field::\protect\T1\textdollarui)
- Fields that are not editable by default will not appear on the form (https://atk4-data.readthedocs.io/en/develop/fields.html#Field::isEditable)
- Field typecasting will be invoked such as for converting dates
- Reference fields (https://atk4-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
- Not-nullable and Required fields will have form controls visually highlighted (https://atk4-data.readthedocs.io/en/develop/fields.html?highlight=required#Field::\protect\T1\textdollarnullable)
- 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 form controls and in which order to display on the form
- Override labels, form control classes
- Group form controls 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 form controls respective fields 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:

- Captcha form control
- File Upload form control (see https://github.com/atk4/filestore)
- Multi-record form

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

5.4.3.2 Layout and Form Controls

Although Form extends the View class, controls are not added into Form directly but rather use a View layout for it in order to create their HTML element. In other words, layout attached to the form is responsible of rendering HTML for fields.

When Form is first initialized, it will provide and set a default Generic layout within the form. Then using *Form::addControl()* will rely on that layout to add form control to it and render it properly. You may also supply your own layout when creating your form.

Form layout may contain sub layouts. Each sub layout being just another layout view, it is possible to nest them, by adding other sub layout to them. This allows for great flexibility on how to place your form controls within Form.

Each sub layout may also contain specific section layout like Accordion, Columns or Tabs.

More on Form layout and sub layout below.

5.4.3.3 Adding Controls

`Atk4\Ui\Form::`**`addControl`**(*$name*, *$decorator = []*, *$field = []*)

Create a new control on a form:

```
$form = Form::addTo($app);
$form->addControl('email');
$form->addControl('gender', [\Atk4\Ui\Form\Control\Dropdown::class, 'values' => ['Female
↪', 'Male']]);
$form->addControl('terms', [], ['type' => 'boolean', 'caption' => 'Agree to Terms &␣
↪Conditions']);
```

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

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

Form control does not have to be added directly into the form. You can use a separate *Form\Layout* or even a regular view. Simply specify property `Form\Control::$form`:

```
$myview = View::addTo($form, ['defaultTemplate' => './mytemplate.html']);
Form\Control\Dropdown::addTo($myview, ['form' => $form]);
```

Adding new controls

First argument to addControl is the name of the form control. You cannot have multiple controls with the same name.

If a field exists 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 control will be added without association with field. This will not work with regular fields, but you can add custom control such as CAPTCHA, which does not really need association with a field.

Form Control

To avoid term miss-use, we use "Field" to refer to `\Atk4\Data\Field`. This class is documented here: https://atk4-data.readthedocs.io/en/develop/fields.html

Form uses a small UI component to visualize HTML input fields associated with the respective Model Field. We call this object "Form Control". All form controls extend from class `Form\Control`.

Agile UI comes with at least the following form controls:

- Input (also extends into Line, Password, Hidden)
- Dropdown
- Checkbox
- Radio
- Calendar
- Radio
- Money

For some examples see: https://ui.atk4.org/demos/form/form3.php

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

```
$form->addControl('accept_terms', [\Atk4\Ui\Form\Control\Checkbox::class]);
$form->addControl('gender', [\Atk4\Ui\Form\Control\Dropdown::class, 'values' => ['Female
↪', 'Male']]);

$calendar = new \Atk4\Ui\Form\Control\Calendar();
$calendar->type = 'tyme';
$calendar->options['ampm'] = true;
$form->addControl('time', $calendar);
```

For more information on default form controls as well as examples on how to create your own see documentation on `Form\Control`.

`Atk4\Ui\Form::`**`controlFactory`**`(`*`\Atk4\Data\Field $field, $defaults = []`*`)`

If form control class is not specified (`null`) then it will be determined from the type of the Data control with `controlFactory` method.

Data Field

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

There are 3 ways to define Data form control using 'string', 'json' or 'object':

```
$form->addControl('accept_terms', [\Atk4\Ui\Form\Control\Checkbox::class], 'Accept Terms␣
↪& Conditions');
$form->addControl('gender', [], ['enum' => ['Female', 'Male']]);

class MyBoolean extends \Atk4\Data\Field
{
    public string $type = 'boolean';
    public ?array $enum = ['N', 'Y'];
```

(continues on next page)

(continued from previous page)

```
}
$form->addControl('test2', [], 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()`. (https://atk4-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:

```
$form = Form::addTo($app);
$form->setEntity(new User($db), []);

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

addControl into Form with Existing Model

If your form is using a model and you add an additional control, then the underlying model field will be created but it will be set as "neverPersist" (https://atk4-data.readthedocs.io/en/develop/fields.html#Field::\protect\T1\textdollarneverPersist).

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

```
class User extends \Atk4\Data\Model
{
    public $table = 'user';

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

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

$form = Form::addTo($app);
$form->setEntity(new User($db));

// add password verification field
$form->addControl('password_verify', [\Atk4\Ui\Form\Control\Password::class], 'Type␣
↪password again');
$form->addControl('accept_terms', [], ['type' => 'boolean']);

// submit event
$form->onSubmit(function (Form $form) {
    if ($form->entity->get('password') != $form->entity->get('password_verify')) {
        return $form->jsError('password_verify', 'Passwords do not match');
```

(continues on next page)

(continued from previous page)

```
    }

    if (!$form->entity->get('accept_terms')) {
        return $form->jsError('accept_terms', 'Read and accept terms');
    }

    $form->entity->save(); // will only store email / password

    return $form->jsSuccess('Thank you. Check your email now');
});
```

Field Type vs Form Control

Sometimes you may wonder - should you pass form control class (Form\Control\Checkbox) or a data field type (['type' => 'boolean']);

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

```
$form = Form::addTo($app);
$form->addControl('date1', [], ['type' => 'date']);
$form->addControl('date2', [\Atk4\Ui\Form\Control\Calendar::class, 'type' => 'date']);

$form->onSubmit(function (Form $form) {
    echo 'date1 = ' . print_r($form->entity->get('date1'), true) . ' and date2 = ' .␣
↪print_r($form->entity->get('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 data type, do not confuse with ui type `=>` date pass as second argument for Calendar field, 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 Form Control from Model

In large projects you most likely won't be setting individual form controls for each Form. Instead you can simply use `setEntity()` to populate all form controls from fields defined inside a model. Form does have a pretty good guess about form control decorator based on the data field type, but what if you want to use a custom decorator?

This is where `$field->ui` comes in (https://atk4-data.readthedocs.io/en/develop/fields.html#Field::\protect\T1\textdollarui).

You can specify `'ui' => ['form' => $decoratorSeed]` when defining your model field inside your Model:

```
class User extends \Atk4\Data\Model
{
    public $table = 'user';

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

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

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

The seed for the UI will be combined with the default overriding `Form\Control\Calendar::$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->addField('birth_year', ['ui' => [\Atk4\Ui\Form\Control\Calendar::class, 'type' =>
↪'month']);
```

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

Using setEntity()

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

property Atk4\Ui\Form::$**model**

Atk4\Ui\Form::**setEntity**(*$model*[, *$fields*])

Associate form controls 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 form controls to import from model fields and their order by simply listing model field names in an array as a second argument.

Specifying "false" or empty array as a second argument will import no model fields as form controls, so you can then use *Form::addControl* to import form controls from model fields individually.

Note that *Form::setEntity* also delegates adding form control to the form layout by using `Form->layout->setEntity()` internally.

See also: https://atk4-data.readthedocs.io/en/develop/fields.html#Field::isEditable

Using setEntity() on a sub layout

You may add form controls to sub layout directly using setEntity method on the sub layout itself.:

```
$form = Form::addTo($app);
$form->setEntity($model, []);

$subLayout = $form->layout->addSubLayout();
$subLayout->setEntity($model, ['first_name', 'last_name']);
```

When using setEntity() on a sub layout to add controls per sub layout instead of entire layout, make sure you pass false as second argument when setting the model on the Form itself, like above. Otherwise all model fields will be automatically added in Forms main layout and you will not be able to add them again in sub-layouts.

Loading Values

Although you can set form control values individually using `$form->entity->set('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->getField('password')->neverPersist = true; // ignore password field
$user = $user->load($currentUserId);

// display all fields (except password) and values
$form = Form::addTo($app);
$form->setEntity($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 fields, unless they're marked as System (https://atk4-data.readthedocs.io/en/develop/fields.html#Field::\protect\T1\textdollarsystem), see https://atk4-data.readthedocs.io/en/develop/model.html?highlight=onlyfields#Model::setOnlyFields:

```
$form = Form::addTo($app);
$form->setEntity((new User($db))->load($currentUserId), ['email', 'name']);
```

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

Validating

The topic of validation in web apps is quite extensive. You should start by reading what Agile Data has to say about validation: https://atk4-data.readthedocs.io/en/develop/persistence.html#validation

Sometimes validation is needed when storing field value inside a model (e.g. setting boolean to "blah") and sometimes validation should be performed only when storing model data into the database.

Here are a 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 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 (https://atk4-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()` (https://atk4-data.readthedocs.io/en/develop/persistence.html#Model::save) not to throw validation exception.
- Form submit handler will also interpret use of *`Form::jsError`* 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';

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

        $this->addField('name', ['required' => true]);
        $this->addField('surname');
        $this->addField('gender', ['enum' => ['M', 'F']]);
    }

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

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

        return $errors;
    }
}
```

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

```
Form::addTo($app)
    ->setEntity(new Person($db));
```

This should display a following form:

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

Form Submit Handling

`Atk4\Ui\Form::`**`onSubmit`**`(`*$callback*`)`

Specify a PHP callback that will be executed on successful form submission.

`Atk4\Ui\Form::`**`jsError`**`(`*$field*, *$message*`)`

Create and return *Js\JsChain* action that will indicate error on a form control.

`Atk4\Ui\Form::`**`jsSuccess`**`(`*$title*[, *$subTitle*]`)`

Create and return *Js\JsChain* action, that will replace form with a success message.

`Atk4\Ui\Form::`**`setApiConfig`**`(`*$config*`)`

Add additional parameters to Fomantic-UI .api function which does the AJAX submission of the form.

For example, if you want the loading overlay at a different HTML element, you can define it with:

```
$form->setApiConfig(['stateContext' => 'my-JQuery-selector']);
```

All available parameters can be found here: https://fomantic-ui.com/behaviors/api.html#/settings

property `Atk4\Ui\Form::$`**`successTemplate`**

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

To continue with the example, a new Person record can be added into the database but only if they have also accepted terms and conditions. An onSubmit handler that would perform the check can be defined displaying error or success messages:

```
$form->onSubmit(function (Form $form) {
    if (!$form->entity->get('terms')) {
        return $form->jsError('terms', 'You must accept terms and conditions');
    }

    $form->entity->save();

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

Callback function can return one or multiple JavaScript actions. Methods such as *Form::jsError()* or *Form::jsSuccess()* will help initialize those actions for your form. Here is a code that can be used to output multiple errors at once. Errors were intentionally not grouped with a message about failure to accept of terms and conditions:

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

    if (!$form->entity->get('name')) {
        $jsErrors[] = $form->jsError('name', 'Name must be specified');
    }

    if (!$form->entity->get('surname')) {
        $jsErrors[] = $form->jsError('surname', 'Surname must be specified');
    }

    if ($jsErrors) {
        return new \Atk4\Ui\Js\JsBlock($jsErrors);
    }
```

(continues on next page)

(continued from previous page)

```
    if (!$form->entity->get('terms')) {
        return $form->jsError('terms', 'You must accept terms and conditions');
    }

    $form->entity->save();

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

So far Agile UI / Agile Data does not come with a validation library but it supports usage of 3rd party validation libraries.

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'))
    ->addMoreInfo('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 and Sub-layout

As stated above, when a Form object is created and form controls are added through either *Form::addControl()* or *Form::setEntity()*, the form controls will appear one under each-other. This arrangement of form controls as well as display of labels and structure around the form controls themselves is not done by a form, but another object - "Form Layout". This object is responsible for the form control flow, presence of labels etc.

`Atk4\Ui\Form::`**`initLayout`**`(`*Form\Layout $layout*`)`

> Sets a custom Form\Layout object for a form. If not specified then form will automatically use Form\Layout class.

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 *Form\Layout* object. You can also specify a header.

Form Control Group Layout and Sub-layout

Controls can be organized in groups, using method `Form::addGroup()` or as sub section using `Form\Layout::addSubLayout()` method.

Using Group

Group will create a sub layout for you where form controls added to the group will be placed side by side in one line and where you can setup specific width for each field.

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

```
$form->setEntity(new User($db), []); // will not populate any form controls automatically

$group = $form->addGroup('Customer');
$group->addControl('name');
$group->addControl('surname');

$group = $form->addGroup('Address');
$group->addControl('street');
$group->addControl('city');
$group->addControl('country');
```

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

```
$group = $form->addGroup('Address');
$group->addControl('address', ['width' => 'twelve']);
$group->addControl('code', ['Post Code', 'width' => 'four']);
```

or you can divide space equally between form controls. Header is also omitted for this group:

```
$group = $form->addGroup(['width' => 'two']);
$group->addControl('city');
$group->addControl('country');
```

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

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

Using Sub-layout

There are four specific sub layout views that you can add to your existing form layout: Generic, Accordion, Tabs and Columns.

Generic sub layout is simply another layout view added to your existing form layout view. You add fields the same way as you would do for *Form\Layout*.

Sub layout section like Accordion, Tabs or Columns will create layout specific section where you can organize fields in either accordion, tabs or columns.

The following example will show how to organize fields using regular sub layout and accordion sections:

```
$form = Form::addTo($app);
$form->setEntity($model, []);
```

(continues on next page)

(continued from previous page)

```
$subLayout = $form->layout->addSubLayout([\Atk4\Ui\Form\Layout\Section::class]);

Header::addTo($subLayout, ['Accordion Section in Form']);
$subLayout->setEntity($model, ['name']);

$accordionLayout = $form->layout->addSubLayout([\Atk4\Ui\Form\Layout\Section\
↪Accordion::class]);

$a1 = $accordionLayout->addSection('Section 1');
$a1->setEntity($model, ['iso', 'iso3']);

$a2 = $accordionLayout->addSection('Section 2');
$a2->setEntity($model, ['numcode', 'phonecode']);
```

In the example above, we first add a Generic sub layout to the existing layout of the form where one form control ('name') is added to this sub layout.

Then we add another layout to the form layout. In this case it's specific Accordion layout. This sub layout is further separated in two accordion sections and form controls are added to each section:

```
$a1->setEntity($model, ['iso', 'iso3']);
$a2->setEntity($model, ['numcode', 'phonecode']);
```

Sub layout gives you greater control on how to display form controls within your form. For more examples on sub layouts please visit demo page: https://ui.atk4.org/demos/form/form-section.php

Fomantic-UI Modifiers

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

```
$form = new \Atk4\Ui\Form(['class.small segment' => true]));
```

For further styling see documentation on *View*.

5.4.3.4 Not-Nullable and Required Fields

ATK Data has two field flags - "nullable" and "required". Because ATK Data works with PHP values, the values are defined like this:

- nullable = value of the field can be null.
- required = value of the field must not be empty/false/zero, null is empty too.

Form changes things slightly, because it does not allow user to enter NULL values. For example - string (or unspecified type) fields will contain empty string if are not entered (""). Form will never set NULL value for them.

When working with other types such as numeric values and dates - empty string is not a valid number (or date) and therefore will be converted to NULL.

So in most cases you'd want "required=true" flag set on your ATK Data fields. For numeric field, if zero must be a permitted entry, use "nullable=false" instead.

5.4.3.5 Conditional Form

`Atk4\Ui\Form::setControlsDisplayRules()`

So far we had to present form with a set of form controls while initializing. Sometimes you would want to hide/display controls while user enters the data.

The logic is based around passing a declarative array:

```
$form = Form::addTo($app);
$form->addControl('phone1');
$form->addControl('phone2');
$form->addControl('phone3');
$form->addControl('phone4');

$form->setControlsDisplayRules([
    'phone2' => ['phone1' => 'notEmpty'],
    'phone3' => ['phone1' => 'notEmpty', 'phone2' => 'notEmpty'],
    'phone4' => ['phone1' => 'notEmpty', 'phone2' => 'notEmpty', 'phone3' => 'notEmpty'],
]);
```

ATK UI relies on rules defined by Fomantic-UI https://fomantic-ui.com/behaviors/form.html, so you can use any of the conditions there.

Here is a more advanced example:

```
$form = Form::addTo($app);
$form->addControl('name');
$form->addControl('subscribe', [\Atk4\Ui\Form\Control\Checkbox::class, 'Subscribe to␣
↪weekly newsletter', 'class.toggle' => true]);
$form->addControl('email');
$form->addControl('gender', [\Atk4\Ui\Form\Control\Radio::class], ['enum' => ['Female',
↪'Male']])->set('Female');
$form->addControl('m_gift', [\Atk4\Ui\Form\Control\Dropdown::class, 'caption' => 'Gift␣
↪for Men', 'values' => ['Beer Glass', 'Swiss Knife']]);
$form->addControl('f_gift', [\Atk4\Ui\Form\Control\Dropdown::class, 'caption' => 'Gift␣
↪for Women', 'values' => ['Wine Glass', 'Lipstick']]);

// show email and gender when subscribe is checked
// show m_gift when gender = 'male' and subscribe is checked
// show f_gift when gender = 'female' and subscribe is checked

$form->setControlsDisplayRules([
    'email' => ['subscribe' => 'checked'],
    'gender' => ['subscribe' => 'checked'],
    'm_gift' => ['gender' => 'isExactly[Male]', 'subscribe' => 'checked'],
    'f_gift' => ['gender' => 'isExactly[Female]', 'subscribe' => 'checked'],
]);
```

You may also define multiple conditions for the form control to be visible if you wrap them inside and array:

```
$form = Form::addTo($app);
$form->addControl('race', [\Atk4\Ui\Form\Control\Line::class]);
$form->addControl('age');
$form->addControl('hair_cut', [\Atk4\Ui\Form\Control\Dropdown::class, 'values' => ['Short
↪', 'Long']]);
```

(continues on next page)

(continued from previous page)

```
// show 'hair_cut' when race contains the word 'poodle' AND age is between 1 and 5
// OR
// show 'hair_cut' when race contains exactly the word 'bichon'
$form->setControlsDisplayRules([
    'hair_cut' => [['race' => 'contains[poodle]', 'age' => 'integer[1..5]'], ['race' =>
↪'isExactly[bichon]']],
]);
```

Hiding / Showing group of field

Instead of defining rules for form controls individually you can hide/show entire group:

```
$form = Form::addTo($app, ['class.segment' => true]);
Label::addTo($form, ['Work on form group too.', 'class.top attached' => true], [
↪'AboveControls']);

$groupBasic = $form->addGroup(['Basic Information']);
$groupBasic->addControl('first_name', ['width' => 'eight']);
$groupBasic->addControl('middle_name', ['width' => 'three']);
$groupBasic->addControl('last_name', ['width' => 'five']);

$form->addControl('dev', [\Atk4\Ui\Form\Control\Checkbox::class, 'caption' => 'I am a␣
↪developer']);

$groupCode = $form->addGroup(['Check all language that apply']);
$groupCode->addControl('php', [\Atk4\Ui\Form\Control\Checkbox::class]);
$groupCode->addControl('js', [\Atk4\Ui\Form\Control\Checkbox::class]);
$groupCode->addControl('html', [\Atk4\Ui\Form\Control\Checkbox::class]);
$groupCode->addControl('css', [\Atk4\Ui\Form\Control\Checkbox::class]);

$groupOther = $form->addGroup(['Others']);
$groupOther->addControl('language', ['width' => 'eight']);
$groupOther->addControl('favorite_pet', ['width' => 'four']);

// to hide-show group simply select a field in that group
// show group where 'php' belong when dev is checked
// show group where 'language' belong when dev is checked

$form->setGroupDisplayRules([
    'php' => ['dev' => 'checked'],
    'language' => ['dev' => 'checked'],
]);
```

class Atk4\Ui**Form\Layout**

Renders HTML outline encasing form controls.

property Atk4\Ui\Form\Layout::$**form**

Form layout objects are always associated with a Form object.

Atk4\Ui\Form\Layout::**addControl**()

Same as *Form::addControl()* but will place a form control inside this specific layout or sub-layout.

5.4.4 Paginator

class Atk4\Ui**Paginator**

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

5.4.4.1 Adding and Using

property Atk4\Ui\Paginator::$**total**

property Atk4\Ui\Paginator::$**page**

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

```
$paginator = Paginator::addTo($app);
$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:

```
$h = Header::addTo($page);
LoremIpsum::addTo($page); // some content here

$p = Paginator::addTo($page);
$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
```

5.4.4.2 Range and Logic

You can configure Paginator through properties.

property Atk4\Ui\Paginator::$**range**

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.

Atk4\Ui\Paginator::**getPaginatorItems**()

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

Atk4\Ui\Paginator::**getCurrentPage**()

Returns number of current page.

5.4.4.3 Template

Paginator uses Fomantic-UI `ui pagination menu` so if you are unhappy with the styling (e.g: active element is not sufficiently highlighted), you should refer to Fomantic-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.

`Atk4\Ui\Paginator::`**`renderItem`**(*$t*, *$page = null*)

5.4.4.4 Dynamic Reloading

`property` `Atk4\Ui\Paginator::$`**`reload`**

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.

5.4.5 Columns

`class` `Atk4\Ui\`**`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\Columns::`**`addColumn`**`()`

When you add new component to the page it will typically consume 100% width of its container. Columns will break down width into chunks that can be used by other elements:

```
$c = Columns::addTo($page);
LoremIpsum::addTo($c->addColumn(), [1]);
LoremIpsum::addTo($c->addColumn(), [1]);
```

By default width is equally divided by columns. You may specify a custom width expressed as fraction of 16:

```
$c = Columns::addTo($page);
LoremIpsum::addTo($c->addColumn(6), [1]);
LoremIpsum::addTo($c->addColumn(10), [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 Fomantic-UI:

```
$c = Columns::addTo($page, ['width' => 4]);
Box::addTo($c->addColumn(), ['red']);
Box::addTo($c->addColumn(['class.right floated' => true]), ['blue']);
```

5.4.5.1 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 control the flow of rows better, you can specify addRow():

```
$c = Columns::addTo($page, ['class.internally celled' => true]);

$r = $c->addRow();
Icon::addTo($r->addColumn([2, 'class.right aligned' => true]), ['huge home']);
LoremIpsum::addTo($r->addColumn(12), [1]);
Icon::addTo($r->addColumn(2), ['huge trash']);

$r = $c->addRow();
Icon::addTo($r->addColumn([2, 'class.right aligned' => true]), ['huge home']);
LoremIpsum::addTo($r->addColumn(12), [1]);
Icon::addTo($r->addColumn(2), ['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 https://fomantic-ui.com/collections/grid.html.

5.4.5.2 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

SIX

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();
```

6.1 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 Actions and Events work.

6.1.1 Actions

An action is represented by a PHP object that can map itself into a JavaScript code. For instance the code for hiding a view can be generated by calling:

```
$jsHide = $view->js()->hide();
```

There are other ways to generate an action, such as using *Js\JsExpression*:

```
$jsAlert = new JsExpression('alert([])', ['Hello world']);
```

Finally, actions can be used inside other actions:

```
$jsAlert = new JsExpression('alert([])', [
    $view->js()->text(),
]);

// will produce alert($('#button-id').text());
```

or:

```
$jsAction = $view->js()->text(new JsExpression('[] + []', [
    5,
    10,
]));
```

All of the above 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.

6.1.2 Events

Agile UI also offers a great way to associate 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 `$jsXxx`.

To execute actions instantly on page load, use `true` as first argument to *View::js()*:

```
$view->js(true, new JsExpression('alert([])', ['Hello world']));
```

You can also combine both forms:

```
$view->js(true)->hide();
```

Finally, you can specify the name of the JavaScript event:

```
$view->js('click')->hide();
```

Agile UI also provides support for an on event binding. This allows to apply events on multiple elements:

```
$buttons = View::addTo($app, ['ui' => 'basic buttons']);

\Atk4\Ui\Button::addTo($buttons, ['One']);
\Atk4\Ui\Button::addTo($buttons, ['Two']);
\Atk4\Ui\Button::addTo($buttons, ['Three']);

$buttons->on('click', '.button')->hide();
```

All the above examples will map themselves into a simple and readable JavaScript code.

6.1.3 Extending

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

- Action can be any arbitrary JavaScript with parameters:
  - parameters are always encoded/escaped,
  - action can contain nested actions,
  - you can build your own integration patterns.
- JsChain provides 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.

6.1.4 Including JS/CSS

Sometimes you need to include an additional .js or .css file for your code to work. See `App:requireJs()` and *App::requireCss()* for details.

6.2 Building actions with JsExpressionable

interface Atk4\Ui**Js\JsExpressionable**

Allow objects of the class implementing this interface to participate in building JavaScript expressions.

Atk4\Ui\Js\JsExpressionable::**jsRender**()

Express object as a string containing valid JavaScript statement or expression.

View class is supported as JsExpression argument natively and will present itself as a valid selector. Example:

```
$frame = new View();
$button->js(true)->appendTo($frame);
```

The resulting Javascript will be:

```
$('#button-id').appendTo('#frame-id');
```

6.2.1 JavaScript Chain Building

class Atk4\Ui**Js\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 these examples we'll be using Jquery which is a descendant of JsChain:

```
$jsChain = new Jquery('#the-box-id');
$jsChain->dropdown();
```

The calls to the chain are stored in the object and can be converted into JavaScript by calling *Js\JsChain::jsRender()*

Atk4\Ui\Js\JsChain::**jsRender**()

Converts actions recorded in JsChain into string of JavaScript code.

Executing:

```
echo $jsChain->jsRender();
```

will output:

```
$('#the-box-id').dropdown();
```

Important: It's considered very bad practice to use jsRender to output JavaScript manually. Agile UI takes care of JavaScript binding and also decides which actions should be available while creating actions for your chain.

`Atk4\Ui\Js\JsChain::_jsEncode()`

JsChain will map all the other methods into JS counterparts while encoding all the arguments using `_jsEncode()`. Although similar to the standard JSON encode function, this method quotes strings using single quotes and recognizes *Js\JsExpressionable* objects and will substitute them with the result of *Js\JsExpressionable::jsRender*. The result will not be escaped and any object implementing *Js\JsExpressionable* interface is responsible for safe JavaScript generation.

The following code is safe:

```
$b = new Button();
$b->js(true)->text($app->getRequestQueryParam('button_text'));
```

Any malicious input through the GET arguments will be encoded as JS string before being included as an argument to `text()`.

6.2.2 View to JS integration

We are not building JavaScript code just for the exercise. Our whole point is ability to link that code between actual views. All views support JavaScript binding through two methods: *View::js()* and *View::on()*.

`Atk4\Ui\View::js([$event[, $otherChain]])`

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 `$otherChain` is specified together with event, it will also be bound to said event. `$otherChain` 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());
```

`Atk4\Ui\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 three buttons and clicking any one will hide it. Only a single action is created:

```
$buttons = View::addTo($app, ['ui' => 'basic buttons']);

\Atk4\Ui\Button::addTo($buttons, ['One']);
\Atk4\Ui\Button::addTo($buttons, ['Two']);
\Atk4\Ui\Button::addTo($buttons, ['Three']);

$buttons->on('click', '.button')->hide();

// generates:
// $('#top-element-id').on('click', '.button', function (event) {
//     event.preventDefault();
//     event.stopPropagation();
//     $(this).hide();
// });
```

View::on() is handy when multiple elements exist inside a view which you want to trigger individually. The best example would be a *Lister* with interactive elements.

You can use both actions together. The next example will allow only one button to be active:

```
$buttons = View::addTo($app, ['ui' => 'basic buttons']);

\Atk4\Ui\Button::addTo($buttons, ['One']);
\Atk4\Ui\Button::addTo($buttons, ['Two']);
\Atk4\Ui\Button::addTo($buttons, ['Three']);

$buttons->on('click', '.button', $b3->js()->hide());

// generates:
// $('#top-element-id').on('click', '.button', function (event) {
//     event.preventDefault();
//     event.stopPropagation();
//     $('#b3-element-id').hide();
// });
```

6.3 JsExpression

class Atk4\Ui**Js\JsExpression**

Atk4\Ui\Js\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:

```
$jsAlert = new JsExpression('alert([])', [
    $view->js()->text(),
]);
```

Because *Js\JsChain* will typically wrap all the arguments through `Js\JsChain::_jsonEncode()`, it prevents you from accidentally injecting 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 plain JavaScript 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!

$js = new JsExpression('document');
echo (new Jquery($js))->find('h1')->hide()->jsRender();

// produces $(document).find('h1').hide();
// works correctly!!
```

6.3.1 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:

```
$jsH1 = $leftBox1->js()->height();
$jsH2 = $leftBox2->js()->height();

$jsSum = new JsExpression('[] + []', [$jsH1, $jsH2]);

$rightBoxContainer->js(true)->height($jsSum);
```

It is important to remember that the 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 lines are identical:

```
$jsSum = new JsExpression('[] + []', [$jsH1, $jsH2]);
$jsSum = new JsExpression('[0] + [1]', [$jsH1, $jsH2]);
$jsSum = new JsExpression('[a] + [b]', ['a' => $jsH1, 'b' => $jsH2]);
```

Important: We have specifically selected a very simple tag format as a reminder not to write any code as part of JsExpression. You must not use JsExpression() for anything complex.

6.3.2 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.

Create a file `test.js` containing:

```
function mySum(arr) {
    return arr.reduce(function (a, b) {
        return a + b;
    }, 0);
}
```

Then load this JavaScript dependency on your page (see `App::includeJS()` and `App::includeCSS()`). Finally use UI code as a "glue" between your routine and the actual View objects. For example, to match the size of `$rightContainer` with the size of `$leftContainer`:

```
$jsHeights = [];
foreach ($leftContainer->elements as $leftBox) {
    $jsHeights[] = $leftBox->js()->height();
}

$rightContainer->js(true)->height(new JsExpression('mySum([])', [$jsHeights]));
```

This will map into the following JavaScript code:

```
$('#right_container_id').height(
    mySum([
        $('#left_box1').height(),
        $('#left_box2').height(),
        $('#left_box3').height(),
        // ...
    ])
);
```

You can further simplify JavaScript code yourself, but keep the JavaScript logic inside the `.js` files and leave PHP only for binding.

6.4 Modals

There are two modal implementations in ATK:

- View - Modal: This works with a pre-existing Div, shows it and can be populated with contents;
- JsModal: This creates an entirely new modal Div and then populates it.

In contrast to *Modal*, the HTML element generated by *Js\JsModal* is always destroyed when the modal is closed instead of only hiding it.

6.4.1 Modal

class Atk4\Ui**Modal**

Atk4\Ui\Modal::**set**(*callback*)

Atk4\Ui\Modal::**jsShow**()

Atk4\Ui\Modal::**jsHide**()

Atk4\Ui\Modal::**addContentClass**()

Atk4\Ui\Modal::**addScrolling**()

Atk4\Ui\Modal::**setOption**()

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

```
$modal = \Atk4\Ui\Modal::addTo($app, ['Modal Title']);
$modal->set(function (View $p) use ($modal) {
    \Atk4\Ui\LoremIpsum::addTo($p);
    \Atk4\Ui\Button::addTo($p, ['Hide'])
        ->on('click', $modal->jsHide());
});

\Atk4\Ui\Button::addTo($app, ['Show'])
    ->on('click', $modal->jsShow());
```

Modal will render as a HTML block but will be hidden. Alternatively you can use Modal without loadable content:

```
$modal = \Atk4\Ui\Modal::addTo($app, ['Modal Title']);
\Atk4\Ui\LoremIpsum::addTo($modal);
\Atk4\Ui\Button::addTo($modal, ['Hide'])
    ->on('click', $modal->jsHide());

\Atk4\Ui\Button::addTo($app, ['Show'])
    ->on('click', $modal->jsShow());
```

The second way is more convenient for creating static content, such as Terms of Service.

You can customize the CSS classes of both header and content section of the modal using the properties `headerClass` or `contentClass` or use the method `addContentClass()`. See the Fomantic-UI modal documentation for further information.

6.4.2 JsModal

class Atk4\Ui\Js**JsModal**

This alternative implementation to *Modal* is convenient for situations when the need to open a dialog box is not known in advance. This class is not a component, but rather an Action so you **must not** add it to the render tree. To accomplish that, use a *VirtualPage*:

```
$vp = \Atk4\Ui\VirtualPage::addTo($app);
\Atk4\Ui\LoremIpsum::addTo($vp, ['size' => 2]);

\Atk4\Ui\Button::addTo($app, ['Dynamic Modal'])
    ->on('click', new \Atk4\Ui\Js\JsModal('My Popup Title', $vp->getUrl('cut')));
```

Note that this element is always destroyed as opposed to *Modal*, where it is only hidden.

Important: See Modals and reloading concerning the intricacies between jsMmodals and callbacks.

6.5 Reloading

class Atk4\Ui**Js\JsReload**

JsReload is a JavaScript action that performs reload of a certain object:

```
$jsReload = new JsReload($table);
```

This action can be used similarly to any other JsExpression. For instance submitting a form can reload some other view:

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

$form = \Atk4\Ui\Form::addTo($app);
$table = \Atk4\Ui\Table::addTo($app);

$form->setEntity($bookModel);

$form->onSubmit(function (Form $form) use ($table) {
    $form->entity->save();

    return new \Atk4\Ui\Js\JsReload($table);
});

$table->setModel($bookModel);
```

In this example, filling out and submitting the form will result in table contents being refreshed using AJAX.

6.5.1 Modals and reloading

Care needs to be taken when attempting to combine the above with a *Js\JsModal* which requires a *VirtualPage* to store its contents. In that case, the order in which declarations are made matters because of the way the render tree is processed.

For example, in order to open a modal dialog containing a form and reload a table located on the main page with the updated data on form submission (thus without having to reload the entire page), the following elements are needed:

- a virtual page containing a JsModal's contents (in this case a form),
- a table showing data on the main page,
- a button that opens the modal in order to add data, and

- the form's callback on submit.

The following will **not** work:

```
$app = new MyApp();
$model = new MyModel();

// JsModal requires its contents to be put into a VirtualPage
$vp = \Atk4\Ui\VirtualPage::addTo($app);
$form = \Atk4\Ui\Form::addTo($vp);
$form->setEntity($model);

$table = \Atk4\Ui\Table::addTo($app);
$table->setModel($model));

$button = \Atk4\Ui\Button::addTo($app, ['Add Item', 'icon' => 'plus']);
$button->on('click', new \Atk4\Ui\Js\JsModal('JSModal Title', $vp));

$form->onSubmit(function (Form $form) use ($table) {
    $form->entity->save();

    return new \Atk4\Ui\Js\JsBlock([
        $table->jsReload(),
        $form->jsSuccess('ok'),
    ]);
});
```

Table needs to be first! The following works:

```
$app = new MyApp();
$model = new MyModel();

// this needs to be first
$table = \Atk4\Ui\Table::addTo($app);
$table->setModel($model));

$vp = \Atk4\Ui\VirtualPage::addTo($app);
$form = \Atk4\Ui\Form::addTo($vp);
$form->setEntity($model);

$button = \Atk4\Ui\Button::addTo($app, ['Add Item', 'icon' => 'plus']);
$button->on('click', new \Atk4\Ui\Js\JsModal('JSModal Title', $vp));

$form->onSubmit(function (Form $form) use ($table) {
    $form->entity->save();

    return new \Atk4\Ui\Js\JsBlock([
        $table->jsReload(),
        $form->jsSuccess('ok'),
    ]);
});
```

The first will not work because of how the render tree is called and because VirtualPage is special. While rendering, if a reload is caught, the rendering process stops and only renders what was asked to be reloaded. Since VirtualPage is special, when asked to be rendered and it gets triggered, rendering stops and only the VirtualPage content is rendered.

To force yourself to put things in order you can write the above like this:

```
$table = \Atk4\Ui\Table::addTo($app);
$table->setModel($model);

$vp = \Atk4\Ui\VirtualPage::addTo($app);
$vp->set(function (\Atk4\Ui\VirtualPage $p) use ($table, $model) {
    $form = \Atk4\Ui\Form::addTo($p);
    $form->setEntity($model);
    $form->onSubmit(function (Form $form) use ($table) {
        $form->entity->save();

        return new \Atk4\Ui\Js\JsBlock([
            $table->jsReload(),
            $form->jsSuccess('ok'),
        ]);
    });
});

$button = \Atk4\Ui\Button::addTo($app, ['Add Item', 'icon' => 'plus']);
$button->on('click', new \Atk4\Ui\Js\JsModal('JSModal Title', $vp));
```

Note that in no case you will be able to render the button *above* the table (because the button needs a reference to `$vp` which references `$table` for reload), so `$button` must be last.

6.6 Background Tasks

Agile UI has addressed one of the big shortcomings of the PHP language: the ability to execute running / background processes. It is best illustrated with an example:

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

The most basic approach is:

```
$button = \Atk4\Ui\Button::addTo($app, ['Process 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 the user was aware of the progress of your process, which is when *Server-Sent Events (JsSse)* comes into play.

6.6.1 Server-Sent Events (JsSse)

class Atk4\Ui**JsSse**

Atk4\Ui\JsSse::**send**(*action*)

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

```
$button = \Atk4\Ui\Button::addTo($app, ['Process Image']);

$sse = \Atk4\Ui\JsSse::addTo($button);

$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

SEVEN

ADVANCED TOPICS

7.1 Agile Data

Agile Data is a business logic and data persistence framework. It's a separate library that has been specifically designed and developed for use in Agile UI.

With Agile Data you can easily connect your UI with your data and make UI components store your data in SQL, NoSQL or RestAPI. On top of the existing persistencies, Agile UI introduces a new persistence class: "UI".

This UI persistence will be extensively used when data needs to be displayed to the user through UI elements or when input must be received from the UI layer.

If you do not intend to store data anywhere or are using your own ORM, the Agile Data will still be used to some extent and therefore it appears as requirement.

Most of the ORMs lack several important features that are necessary for UI framework design:

- ability to load/store data safely with conditions.
- built-in support for column meta-information
- field, type and table mapping
- "onlyFields" support for efficient querying
- domain-level model references.

Agile Data is distributed under same open-source license as Agile UI and the rest of this documentation will assume you are using Agile Data for the purpose of overall clarity.

7.2 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.

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

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

7.3.1 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.

7.3.2 Business Logic Unit Tests

Those tests are most suitable for testing your business logic, that is included in Agile Data. Use "array" persistence to pre-set model with the necessary data, execute your business logic with mock objects.

1. set up mock database arrays
2. instantiate model(s)
3. execute business operation
4. assert new content of array.

In most cases the Integration tests are easier to make, and give you equal testability.

7.3.3 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 invoice
4. assert through other models e.g. by running client report model.

7.3.4 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.

7.3.5 User Testing

Once you place your components on your pages and associate them with your actual data you can perform user tests.

CHAPTER

EIGHT

INDICES AND TABLES

- genindex
- search

PHP NAMESPACE INDEX

a

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

T

U

V

W