ÁTOMO FRAMEWORK - DOCS

Introdução

Desenvolvido originalmente para tratar tarefas rotineiras relacionadas ao desenvolvimento web, que vão desde gerenciamento de interface, banco de dados a restful api's, o Atomo Framework une uma gama de mini-ferramentas que foram unificadas em um só, formando esse ecossistema simples, porém poderoso, que agiliza certos processo do dia-a-dia.

Características Principais:

- Modularizado: componentes independentes que consomem o mesmo core
- Suporte nativo a Mysql: contudo é possível configurá-lo em outras engines, desde que seja SQL;
- Configurações Apache: Reescrita de url e proteção de diretórios
- Customizável: Com um domínio intermediário de PHP OO você consegue manipular o core do framework da forma que quiser.
- **base applications**: são arquivos de base que funcionam como linha de frente da aplicação, podendo conter mais de um base application em uma aplicação para trabalhar com múltiplos layouts.

Outro projeto similar está sendo desenvolvido e se encontra em construção de core, também outro framework que muda um pouco a abordagem de trabalho do átomo, este se chama **PHP Factory**, ainda sem data de lançamento oficial. Ambos os projetos serão mantidos e aprimorados, sua licença é livre e estarão disponíveis no repositório oficial do github junto com a documentação oficial

Estrutura de Pastas

- **app**
  application.php
- **core**
  - **src** [arquivos de core]
    .htaccess
- **global**
  - **src** [arquivos globais]
- **config**
  - .htaccess
  - config.atomo
- **modules**
  - **docs**
    - **controllers**
    - **models**
    - **views**
    - validate.php
    - helpers.php
    - config.module
  - .htaccess
- **storage**
- **public**
  - packages.json
  - **theme-app**
    - **css**
    - **js**

observação: Os arquivos .htaccess são arquivos de configuração apache, ou seja, eles modificam valores padrões do próprio servidor para ocasionar uma ação diferente. no caso do .htaccess da raiz do projeto, ele trabalha na reescrita de url, possibilitando as “urls amigáveis" que são unanimidade nos frameworks atuais, e os .htaccess que estão espalhados nos diretórios fora da raiz são para controle e proteção de arquivos, uma vez que por url, não será possível listar os arquivos do diretório do framework enquanto o htaccess estiver funcionando.

APP

Diretório que armazena os arquivos bases de aplicação, cada módulo seta um arquivo aplicação para mirar, usando a estrutura de pastas como exemplo, poderíamos dizer que o módulo docs possui em seu config.module uma chamada ao app application.php, vejamos no exemplo abaixo:

~ arquivo: config.module

```
-------------------------------------
name=docs - documentation
version=1.0.0
app=application
-------------------------------------
```

Um arquivo de aplicação é a base de cada módulo, ou módulos, ou seja, um arquivo html que irá carregar as chamadas de view, css, js, inclusão de componentes com menu, rodapé e entre outros pedaços de código que eventualmente irão se repetir constantemente.

no exemplo acima, podemos determinar que o módulo **docs** está fazendo uma ligação direta com o app **application**(app/application.php), sendo assim, sempre que este módulo for requisitado, o app carregado será **application.php**

CORE

Diretório raíz do framework, onde são feitas as validações internas primárias para funcionamento das regras aplicadas pelo desenvolvedor ou regras predefinidas do próprio sistema como por exemplo: carregamento de módulos, leitura de arquivos .átomo, leitura de arquivos .module, controle de acesso de usuário, rotas, template e afins.

Obs: caso você não conheça a fundo a linguagem PHP, não recomendo que modifique esses arquivos, em caso de customização e ciência do que está querendo fazer, recomendo que leia a documentação do core do átomo framework.

GLOBAL

Diretório destinado para criação de arquivos que serão carregados junto ao **autoload** de core, ou seja, eles sempre estarão disponíveis em qualquer parte da aplicação que possua o autoload. Como exemplo de arquivo global que poderia ser aplicado temos um Helper.php, uma classe global que possui vários métodos estáticos (ou seja, não necessita instância de objeto) e como sendo uma classe global, pode ser usada em qualquer local da aplicação sem restrições.

CONFIG

Como o próprio nome já sugere, este diretório armazena as configurações gerais da aplicação, sendo que uma delas é um .htaccess que apenas protege o diretório enquanto o outro, que mais importa, é o **config.atomo**, nele estão as informações de sistema.

exemplo ~ config.atomo:

> **folder**={ /diretório no qual o sistema está, vazio se for na raiz }
> **listen**={ se ele esta sobre alguma porta }
> **version**={ versão da sua aplicação }
> **protocol**={ protocolo web [http ou https] }
> **debug**={ habilitar ou desabilitar erros / 0=off , 1=on }
> **login_access_status**={ se havera controle de sessão, 0=off, 1=on }
> **login_access_name**={ nome padrão do usuario na sessão }
> **login_access_module**={ nome do modulo de autenticação/login }
> **login_access_view**={ view/rota padrão de autenticação }
> **default_module**={ modulo padrão da aplicação/ primeiro a carregar }
> **engine_db**={ engine sql padrão, no caso, o átomo usa db-sql }

STORAGE

Diretório destinado aos arquivos que serão armazenados em disco no decorrer do funcionamento do sistema, sejam eles, uploads, logs, arquivos temporários, arquivos de sistema e afins.

MODULES

Diretório cujo a maior parte do desenvolvimento é feita, onde se localizam os módulos da aplicação Um módulo apenas é válido quando o mesmo possui as 5 propriedades do módulo átomo:

1. /**controllers**: diretório das controllers do módulo
2. /**models**: diretório onde serão desenvolvidos as classes do sistema
3. /**views**: diretório das telas do sistema, onde ficarão as views(com extensão .php) que serão exibidas dependendo da rota do módulo
4. **validate.php**: arquivo que controla as rotas, erros, sessões e o que mais for necessário para o funcionamento correto do módulo
5. **config.module**: Arquivo de configuração do módulo, onde será setado, seu nome, sua versão e outras propriedades.

PUBLIC

Diretório destinado a(aos) template(s) da aplicação.
ex:

/public
----- /**admin-lte** [diretório raiz do tema/template]
---------- / **assets**
--------------- / **css**
--------------- style-login.css
--------------- style-sistema.css
--------------- / **js**
--------------- script-login.js
--------------- script-sistema.js

Outro arquivo de extrema importância é o **packages.json,** ele possibilita o desenvolvedor criar **pacotes** de css e js que serão inseridos la em /app/application.php pela engine do átomo, os pacotes são chamados nos arquivos **validate.php** dos módulos, sendo possivel carregar mais de um pacote de arquivos para carregar no **application.php**
segue abaixo um exemplo:

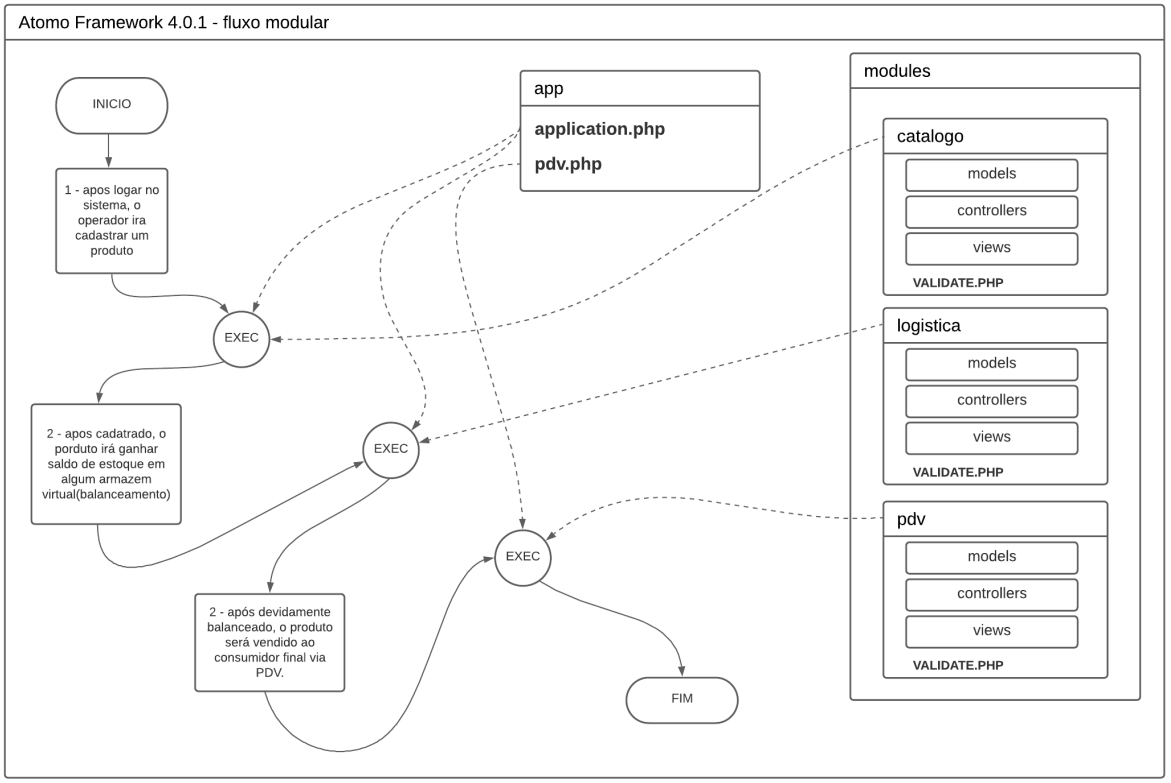
~ exemplo: packages.json

```
{
    "admin-login": {
        "css": [
            "public/admin-lte/css/style-login.css",
        ],
        "js": [
            "public/admin-lte/js/script-login.js",
        ]
    },
    "admin-sistema": {
        "css": [
            "public/admin-lte/css/style-sistema.css",
        ],
        "js": [
            "public/admin-lte/js/script-sistema.js",
        ]
    }
}
```

FLUXO DE APLICAÇÃO

O exemplo abaixo retrata um simples fluxo de 3 etapas utilizando 3 **módulos** e 2 **base applications**, desempenhando um processo bastante comum nos sistemas ERP que é o cadastro, balanço e venda de um produto

Seguindo o padrão do fluxograma convencional, temos um fluxo delimitado entre **INÍCIO** e **FIM**, contudo, as setas sólidas representam o andamento do fluxo enquanto as setas pontilhadas indicam importações de dependências.



O core realiza a leitura da **URI** e faz o carregamento do módulo chamado junto com suas **view** que são as telas, suas **controllers** que vão realizar a ação de cadastro e suas **models** que vão pré-processar os dados para entrada em um banco de dados, findando assim a estrutura **MVC**.

Observando a etapa 1 do fluxo vemos que a requisição solicita o carregamento do módulo **catalogo** e de uma view que irá carregar um formulario, o core realiza a leitura do arquivo **config.module** do módulo e chama a **base application** que está vinculada ao módulo, que no caso é o arquivo **application.php.** O mesmo processo é realizado na etapa 2 e na etapa 3, sendo que na etapa 3 o **base application** é diferente, pelo fato de que o PDV requerer um layout diferente.

PADRÕES DE CÓDIGO - MÉTODOS E FUNÇÕES

ATOM

A classe Atom fornece as funções primárias do kit de desenvolvimento átomo, podendo ser instanciada ou estendida por outras classes modulares ou globais.

Método **path()**

- parâmetros: nenhum
- estático: não
- acesso: público
- retorno: string

retorna o caminho do servidor a nível de disco desde sua raiz (/var ou C:/) até o diretório da aplicação.

exemplo com instância:

```
$app = new Atom();
echo $app->path();
// vai imprimir /var/www/html/pasta_do_projeto
```

exemplo com herança:

```
class Arquivo extends Atom
{
	public function __construct()
	{
		echo parent::path();
		// também vai imprimir /var/www/html/pasta_do_projeto
	}
}
```

Método **index()**

- parâmetros: nenhum
- estático: não
- acesso: público
- retorno: **STRING**

retorna o link de acesso via http ou https da raiz da aplicação.

exemplo com instância:

```
$app = new Atom();
echo $app->index();
// vai imprimir https://localhost/pasta_do_projeto
```

exemplo com herança:

```
class Arquivo extends Atom
{
	public function __construct()
	{
		echo parent::path();
		// vai imprimir https://localhost/pasta_do_projeto
	}
}
```

Método **uri(**$param1**)**

- parâmetro 1: (int) posição do uri na url (não obrigatório)
- estático: não
- acesso: público
- retorno: **ARRAY STRING** ou **STRING**

retorna o valor correspondente ao parâmetro da uri
obs: por padrão o átomo valida o **módulo** como primeiro valor da uri e a **view** do módulo como segundo.

ex: **http://sistema.com/modulo/tela**
então: modulo = 1 e tela = 2

ex: **http://sistema.com/modulo/tela/id/nome**
então: modulo = 1, tela = 2, id = 3 e nome = 4

exemplo com instância:
considerando a url: http://meusistema.com/modulo/tela/listar

```
$app = new Atom();
echo $app->uri(1);
// vai imprimir: modulo
echo $app->uri(2);
// vai imprimir: tela
echo $app->uri(3);
// vai imprimir: listar
```

exemplo com herança:

```
class Validar extends Atom
{
	public function __construct()
	{
		echo parent::uri(3);
		// vai imprimir: listar
	}
}
```

obs

considerando a url: ***http://meusistema.com/modulo/tela/listar***

Se uri() for chamado sem passar nenhum parâmetro ele recebe todos os parâmetros da uri em forma de array.

exemplo:

```
$app = new Atom();
var_dump($app->uri());
/*
	vai imprimir:
	Array(
		0 = ‘modulo’
		1 = ‘tela’
		2 = ‘listar’
	)
	obs: as chaves do array representam a posição dos valores
	de uri() considerando chave + 1, ou seja, modulo = 1, tela = 2
	 e listar = 3
*/
```

Se o parâmetro passado em uri() não for encontrado, ou seja, o parâmetro que representa uma chave posicional de determinado elemento não corresponder ao elemento visado, o retorno é **false**.

Considerando a mesma url do topo desta pagina:

```
$app = new Atom();
echo $app->uri(3);
// vai imprimir: listar
echo $app->uri(4);
vai imprimir: 0,false ou simplesmente NADA, já que seu retorno é false
```

Método **view($param1, $param2)**

- parâmetro 1 = (string) nome da view
  parâmetro 2 = (string) módulo (não obrigatório)
- estático: não
- acesso: público
- retorno: **STRING**

retorna o **link** de acesso a uma **view** de determinado módulo.

exemplo com instância:

```
$app = new Atom();
echo $app->view('listar-pedidos','pedidos');
// vai imprimir: vai imprimir: https://sistema.com/pedidos/listar-pedidos
echo $app->view('detalhes-do-pedido');
/*
	considerando que o atual módulo que a aplicação está carregando ou o
	módulo padrão da aplicação seja pedidos
	vai imprimir: https://sistema.com/pedidos/listar/pedidos
*/
```

exemplo com herança:

```
class Validar extends Atom
{
	public function __construct()
	{
		$linkView = parent::view('detalhes-do-pedido');
		/*
			vai imprimir:
			https://sistema.com/pedidos/listar/pedidos
		*/
	}
}
```

Método **controller($param1, $param2)**

- parâmetro: 1 = (string) nome do arquivo controller
  parâmetro: 2 = (string) módulo (não obrigatório)
- estático: não
- acesso: público
- retorno: **STRING**

retorna o link de acesso a uma controller de determinado módulo.

exemplo com instância:

```
$app = new Atom();
echo $app->controller('deletar-usuario','usuarios');
/*
	vai imprimir:
	https://sistema.com/modules/usuarios/controllers/deletar-usuario.php
*/
```

exemplo com herança:

```
class Rotas extends Atom
{
	public function __construct()
	{
		echo parent::controller('deletar-usuario','usuarios');
		/*
			vai imprimir:
			https://sistema.com/modules/usuarios/controllers/deletar-usuario.php
		*/
	}
}
```

SESSION

A classe **Session** é responsável pelo tratamento das sessões geradas no servidor, seja ela acesso de usuário ou apenas cache.

Método **new($param1, $param2)**

- parâmetro 1 = dados da nova sessão
  parâmetro 2 = nome da sessão (não obrigatório)
- estático: sim
- acesso: público
- retorno: **ARRAY STRING**

cria uma sessão utilizando os parâmetros setados no servidor

exemplo:

```
$dataSession = [
        'idade' => 23,
        'profissao' => 'analista de sistemas'
];

$newSession = Session::new($dataSession);
print_r($newSession);

/* vai imprimir:
        Array(
                name => 'ff341f57a08e0e4f80fcd980e4efe0da39989b67'
        )
*/
```

observações:

*1 - Caso o nome da sessão não seja determinado na sua criação (segundo parâmetro do método new()), o atomo utiliza o valor '***login_access_name***' do arquivo de configurações gerais config.atomo em core/config/config.atomo.*

2 - O nome da sessão é criptografado utilizando o algoritmo ***sha1***

Método **auth($param1)**

- parâmetro 1 = nome da sessão (não obrigatório)
- estático: sim
- acesso: público
- retorno: **BOOLEANO**

verifica se a sessão setada como parâmetro existe e se ela é valida, caso não seja passado nenhum parâmetro, a sessão padrão é usada na validação.

exemplo:

```
class Login {

	public function autenticacao()
	{
		if( Session::auth() ){
			echo 'Usuário já está logado...';
			// imprime: Usuário já esta logado...
		}
	}
}
```

Método **get($param1)**

- parâmetro 1 = nome da sessão (não obrigatório)
- estático: sim
- acesso: público
- retorno: **VARIADO**

Retorna os dados da sessão baseados no nome, caso o nome da sessão não seja setado, é retornado os dados da sessão padrão.
Caso a sessão não exista, o retorno é **false**.

```
print_r( Session::get('carrinho') );

/*
	vai imprimir:
	Array(
		'id' = 1,
		'nome' = 'bolsa termica'
		'valor' = 188.9
	)
*/
```

Método **edit($param1, $param2)**

- parâmetro 1 = (variado) novo valor da sessão
  parâmetro 2 = nome da sessão (não obrigatório)
- estático: sim
- acesso: público
- retorno: **BOOLEANO**

Modifica os valores da sessão.

```
/*
        considerando que já existe uma sessão de carrinho cujo possua os
        produtos listados abaixo em um array:

        Array(
                0 => Array(
                        'id' => 1,
                        'nome' => 'bolsa termica'
                        'valor' => 188.9,
                        'quantidade' => 5
                ),
                1 => Array(
                        'id' => 2,
                        'nome' => 'chapéu',
                        'valor' => 14.5,
                        'quantidade' => 4
                )
        )
*/

class Carrinho
{
        public function quantidadeItem($quantidade, $chave)
        {
                // obtendo todo o carrinho
                $carrinho = Session::get('carrinho');
                // aplicando nova quantidade na chave específica
                // se a chave for 1, o item editado será chapeu
                $carrinho[$chave]['quantidade] = $quantidade;
                // salvando novo array no carrinho da sessão
                $salvar = Session::edit($carrinho,'carrinho');
                // $salvar podera ser TRUE ou FALSE
        }
}
```

Método **done($param)**

- parâmetro = nome da sessão (não obrigatório)
- estático: sim
- acesso: público
- retorno: **NULO**

Deleta a sessão cujo o nome for setado, caso não seja setado nenhum nome, o átomo encerra todas as sessões.

```
class Carrinho
{
	public function limpar()
	{
		// limpando a sessão carrinho
		Session::done('carrinho');
		// verificando se foi limpo
		if( Session::auth('carrinho') == false ){
			echo 'carrinho limpo';
			// imprime: carrinho limpo
		}
	}
}
```

TENGINE

A classe **TEngine** é responsável pela manipulação dos arquivos **base application** e carregamento de dependências gerais.

segue abaixo um exemplo de **base application** configurado com TEngine localizado em **/app**

```php
<?php
  $template = new TEngine();
?>
<!DOCTYPE html>
<html lang="pt-br">
    <head>
        <title><?=$template->title(); ?></title>
        <?php
            $template->css();
        ?>
    </head>
    <body>
        <?php
            // importando o componente menu do
            // projeto para uso no template
            $template->component("menu");
            // carregando a view que foi carregada
            // na validate.php do modulo carregado
            $template->invokeView();

            // carregando o rodapé do projeto
            $template->component("rodape");

            // carregando as chamadas javascript
            $template->js();
        ?>
    </body>
</html>
```

Método **set($param)**

- parâmetro = (string array) dados da view que a template engine consome
- estático: não
- acesso: público
- retorno: **NULO**

Determina quais informações serão carregadas no **base application** baseado na requisição de determinada view. seu uso é feito no arquivo **validate.php** de cada módulo para otimização no controle das rotas.

exemplo:

link requisitado: sistema.com/gestao/**listar-usuarios**

```
$template = new TEngine();
$template->set([
        ‘listar-usuarios’ => [
                'title'   => Atomo|| listar usuários do sistema',
                'file'      => ‘list-users’,
                'packages' => [bootstrap],
                "access" => true
        ]
  ]);
```

o código acima determina que ao ser requisitada a view **listar-usuarios** do módulo gestão, será configurada uma rota determinando as seguintes propriedades:

1. **title**: O título na página que será carregado em **<title></title>** no HTML do **base application**.
2. **file**: Determina qual arquivo em **/{módulo}/views/** será carregado para aquela rota em questão (obs: sem uso da extensão .php no fim).
3. **packages**: um array que determina quais serão os pacotes chamado quando a rota for acionada, os pacotes são definidos em **/public/packages.json**
4. **access**: flag que determina se o acesso da rota está liberado ou não para acesso.

Método **error($param)**

- parâmetro = (string array) dados da view que a template engine consome
- estático: não
- acesso: público
- retorno: **NULO**

Configura um erro de requisição

exemplo:

```
$template = new TEngine();
$template->set([
        ‘404’ => [
                'title'   => ‘Erro 404 - arquivo não encontrado',
                'file'      => ’not-found’,
                'packages' => [bootstrap],
                "access" => true
        ],
        ‘500’ => [
                'title'   => ‘Erro 500 - Erro interno de servidor',
                'file'      => ‘server-error’,
                'packages' => [bootstrap],
                "access" => true
        ]
  ]);
```

Método **invokeView()**

- parâmetro = **NULO**
- estático: não
- acesso: público
- retorno: **NULO**

Carrega a view no arquivo **base application** determinada pelo controle de rotas de **validate.php** na chave file do método **set()** ou **error()**.

exemplo:

```
$template = new TEngine();
$template->invokeView();
```

Método **component($param)**

- parâmetro = nome do componente a ser importado
- estático: não
- acesso: público
- retorno: **NULO**

Carrega arquivos de componentes view localizados em **/app/components**, método similar ao **include/include_once**.

exemplo, baseado na existência de um arquivo **menu.php** localizado em **app/components**:

```
$template = new TEngine();
$template->component('menu');
```

Método **title()**

- parâmetro = **NULO**
- estático: não
- acesso: público
- retorno: **NULO**

Carrega na tag **title**(HTML) do arquivo **base application** o título da página determinado pelo controle de rotas na **validate.php**.

exemplo:

```
<?php
	$template = new TEngine();
?>
<html>
	<head>
		<meta charset="utf-8" />
		<title><?=$template->invokeView(); ?></title>
	</head>
	...
```

Método **css()**

- parâmetro = **NULO**
- estático: não
- acesso: público
- retorno: **NULO**

Carrega os arquivos e **CDN's** CSS nas tags **link**(HTML) no arquivo **base application** baseado nos pacotes determinado pelo controle de rotas em **validate.php**.
Os pacotes de css são criados no arquivo de controle **packages.json** em **/public**.

exemplo:

```
<?php
        $template = new TEngine();
?>
<html>
        <head>
                <meta charset="utf-8" />
                <title><?=$template->invokeView(); ?></title>

                <!-- carrega todos os css's e suas tags <link /> -->
                <?=$template->css(); ?>
        </head>
        ...
```

Método **css()**

- parâmetro = **NULO**
- estático: não
- acesso: público
- retorno: **NULO**

Carrega os arquivos e **CDN's** javascript nas tags **script**(HTML) no arquivo **base application** baseado nos pacotes determinado pelo controle de rotas em **validate.php**.

Os pacotes de js são criados no arquivo de controle **packages.json** em **/public**.

exemplo:

```
<?php
	$template = new TEngine();
?>
<html>
	<head>
		<meta charset=”utf-8” />
		<title><?=$template->invokeView(); ?></title>

		<!-- carrega todos os css’s e suas tags <link /> -->
		<?=$template->css(); ?>
	</head>

	<body>
		<!-- carrega todos os js e suas tags <link /> -->
		<?=$template->js(); ?>
	</body>
</html>.
```