UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
CURSO DE BACHARELADO EM SISTEMAS DE INFORMAÇÃO

Evandro Mazuco de Souza
Lucas Pacheco Teixeira

# SEPTO FRAMEWORK

Uma ferramenta voltada para o desenvolvimento de sistemas web com PHP utilizando Zend Framework e o Doctrine numa arquitetura de três camadas

Florianópolis, SC
2009

Evandro Mazuco de Souza
Lucas Pacheco Teixeira

# SEPTO FRAMEWORK

Uma ferramenta voltada para o desenvolvimento de sistemas web com PHP utilizando Zend Framework e o Doctrine através de uma arquitetura de três camadas

Trabalho de Conclusão de Curso apresentado como exigência parcial para a obtenção do Diploma de Gradução em Sistemas de Informação da Universidade Federal de Santa Catarina

Orientador: Prof. Antônio Carlos Mariani

Co-orientador: Prof. Ricardo Pereira e Silva

Florianópolis, SC
2009

Evandro Mazuco de Souza
Lucas Pacheco Teixeira

**SEPTO FRAMEWORK**

Uma ferramenta voltada para o desenvolvimento de sistemas web com PHP utilizando Zend Framework e o Doctrine através de uma arquitetura de três camadas

Aprovado em:

ORIENTADORES:

_____
Prof. Antônio Carlos Mariani

_____
Prof. Ricardo Pereira e Silva

BANCA EXAMINADORA:

_____
Prof. José Eduardo De Lucca

_____
Profa. Maria Marta Leite

## RESUMO

Este trabalho apresenta o Septo Framework, uma ferramenta que utiliza o Zend Framework e o Doctrine para auxiliar o processo de desenvolvimento de sistemas web através da uma arquitetura de três camadas: apresentação, negócio e persistência.

O trabalho também apresenta os Septo Modules, módulos especiais que podem ser utilizados junto com o Septo Framework para facilitar o desenvolvimento, a manutenção (administração) e a utilização dos sistemas.



## ABSTRACT

This paper presents the Septo Framework, a tool that uses the Zend Framework and Doctrine to assist in development of web systems through a three-tier architecture: presentation, business and persistence.

The paper also shows the Septo Modules, special modules that can be used together with the Septo Framework in order to smooth the development, maintenance (managment) and usage of the systems.

# SUMÁRIO

# LISTAS DE SIGLAS E ABREVIATURAS

**MVC** - Model View Controller

**ORM** - Object-relational Mapping

**SF** - Septo Framework

**ZF** - Zend Framework

# 1. INTRODUÇÃO

PHP é uma linguagem de computador criada por Rasmus Lerdorf em 1995. Tem como características o fato de ser uma linguagem interpretada, livre e amplamente utilizada em servidores web para gerar conteúdo dinâmico. Segundo a Tiobe Software[1], PHP é a 5ª linguagem de programação mais utilizada no mundo, estando a frente de linguagens como Python, C# e Ruby. Fica atrás apenas de Java, C, C++ e Basic, nessa ordem.

Estima-se que existam mais de 20 milhões de web sites que utilizem PHP e isso representa 30% de *market share*[2]. É importante salientar que PHP foi durante muito tempo uma linguagem de *script* voltada exclusivamente para o lado servidor. Sua versão *desktop*, o PHP GTK, é mais recente e muito menos disseminada. Somando-se a isso, ainda existe um outro fator que fortalece a linguagem PHP que são as vantagens oferecidas pelos vários *frameworks* que surgiram mais recentemente para facilitar o desenvolvimento nesta linguagem. CakePHP, Symfony, Prado, Smarty, Propel, Doctrine e Zend Framework são alguns exemplos.

Diante destas vantagens relacionadas à linguagem PHP, sempre foi bastante frustrante observar o descompasso entre teoria e prática relacionado ao desenvolvimento de sistemas utilizando esta linguagem.

Durante a realização desta graduação nos sentimos motivados em aplicar, sempre que possível, padrões de projetos adequados nos sistemas desenvolvidos ao longo dos últimos anos. Por outro lado, na prática, parecia impossível fugir de emaranhados de código que impediam o crescimento organizado dos sistemas e tornavam a sua manutenção uma tarefa desgastante.

Diante de tal cenário e com muita vontade de programar teve início uma tarefa que já dura mais de 2 anos. O resultado desse esforço foi a criação da ferramenta que apresentaremos neste TCC, o Septo Framework, e um ganho

---

[1] Tiobe Software na web: http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html
[2] Zend Technologies, An Overview on PHP

substancial de experiência em desenvolvimento de sistemas em PHP utilizando o Zend Framework[3] e o Doctrine[4].

O Septo Framework utiliza uma arquitetura de três camadas – Apresentação, Negócio e Persistência – seguindo o padrão de projeto *Tree layers architecture*. Na camada de apresentação utiliza principalmente o Zend Framework e, na camada de persistência, o Doctrine.

O Septo Framework utiliza, portanto, estes dois frameworks, Zend e Doctrine, unido-os em uma mesma ferramenta cuja função, além de integrá-los, é oferecer uma série recursos com a intenção de tornar o processo de desenvolvimento de nossos sistemas web em PHP simples, organizado e fáceis de manter. Tal esforço pode ainda significar vantagens para outros times de desenvolvedores interessados em trabalhar seguindo alguns padrões de projetos, sobretudo para aqueles que já conhecem Zend Framework e Doctrine.

O Septo Framework surge para integrar ambos os *frameworks* e vai além, propõe uma arquitetura em três camadas e oferece ferramentas especiais - **Septo Modules -** para automatizar parte do processo de desenvolvimento de sistemas, tornando mais rápido a criação de sistemas completos sob esta arquitetura.

---

[3] Framework PHP que, dentre outras características, implementa o padrão de projeto MVC, Model-View-Controller.
[4] Framework ORM – Object-Relacional Mapping – para PHP.

# 2. OBJETIVOS

## 2.1 Objetivo Geral

Desenvolver o **Septo Framework**, um *framework* para PHP baseado numa arquitetura de três camadas – Apresentação, Negócio e Persistência – que especializa e integra outros dois *frameworks* para PHP, o **Zend Framework** e o **Doctrine**.

## 2.2 Objetivos Específicos

1. Integrar o Zend Framework e o Doctrine em uma mesma ferramenta pronta para ser utilizada;
2. Validar o Septo Framework e os Septo Modules através da elaboração de um sistema que utiliza a ferramenta proposta;

# 3. ZEND FRAMEWORK

## 3.1 Visão geral

O Zend Framework é uma ferramenta implementada em PHP5 e distribuída sob uma licença *New BSD License*. Seu desenvolvedor e mantenedor oficial é a Zend Technologies, uma empresa co-fundada por Andi Gutmans e Zeev Suraski, co-criadores da linguagem PHP.

O ZF tem como foco a construção de aplicações seguras, confiáveis e modernas e faz isso oferecendo uma vasta coleção de componentes voltados para finalidades que incluem as necessidades mais comuns dos desenvolvedores web tais como: controle de acesso, autenticação de usuários, localização, validações, construção de formulários, persistência e vários outros.

Além disso, o ZF oferece em sua biblioteca diversas classes consumidoras de *web services* populares como: Google, Amazon, Yahoo! e Flickr.

## 3.2 O MVC do ZF

A implementação MVC do Zend Framework é uma de suas partes mais interessantes. Isso torna a criação de sistemas web muito mais organizada e facilita a manutenção. Nesta parte do trabalho, vamos compreender como que o padrão MVC foi implementado no Zend Framework e preparar o terreno para entender como o Septo Framework beneficia-se dele.

Basicamente, para que o MVC do Zend Framework entre em ação, é necessário um outro padrão de projeto chamado *Front-Controller*. Por sua vez, para que o *Front-Controller* seja carregado, é necessário a criação de um arquivo de *bootstrap*. Esse arquivo de bootstrap geralmente é um arquivo chamado **index.php** localizado na pasta pública do site cuja função, dentre outras coisas, é carregar e chamar o *Front-Controller* do Zend, uma instância de **Zend_Controller_Front** e mandá-lo trabalhar.

O trabalho principal do **Zend_Controller_Front** é encontrar, com base na *url* solicitada, qual *action* chamar. Uma *action*, para o Zend Framework, é um método público que não retorna nada em alguma instância de um

**Zend_Controller_Action**. Logo em seguida, daremos um exemplo prático e tudo ficará mais claro de entender.

Uma vez encontrado a *action* para uma determinada solicitação, posso invocar a camada de negócio, carregar algumas entidades e colocar os resultados em uma instância de **Zend_View**. Após executar a *action*, automaticamente o Zend Framework irá dar continuidade ao atendimento da solicitação tratando de renderizar a *view*.

Para isso, buscará o *script* correspondente àquela *action* e, uma vez encontrado o *script*, irá executá-lo para gerar uma resposta completa. Neste ponto, aqueles valores que foram colocados na *view* enquanto ainda estávamos dentro da *action* poderão agora ser utilizados aqui, por exemplo, para criar uma tabela html listando um série de clientes de uma empresa.

Tudo isso será abordado em detalhes neste capítulo e vamos começar mostrando o processo de instalação da ferramenta Zend Framework em nossa aplicação.

## 3.3 Instalando o Zend Framework

### *3.3.1 A Estrutura Básica de Diretórios e o Arquivo de Reescrita de URLs*

Por experiência própria, fazer o Zend rodar é o degrau mais alto a ser superado no sentido de dominá-lo. Não é um processo complicado, mas exige o domínio de alguns aspectos que podem dificultar a sua utilização. Após passar bastante tempo testando diferentes formas de carregar o Zend e utilizar sua implementação MVC, nos sentimos motivados a apresentar uma forma didática de começar a utilizá-lo.

A primeira coisa a fazer é o *download* do Zend Framework. Para isso, basta ir até o seu site (*framework.zend.com.br)* e escolher entre as versões *Full Package* ou *Minimal Package*. No momento em que este trabalho é escrito, o Zend Framework encontra-se na sua versão 1.9.2. Para este exemplo de inicialização prático vamos escolher o *Minimal Package* contendo apenas a pasta com código fonte do framework sem oferecer nenhuma aplicação de exemplo.

Uma vez baixado o arquivo que possui em torno de 4MB devemos descompactá-lo e, a partir desse ponto, podemos nos preocupar em criar a estrutura de diretórios necessária para toda a aplicação.

Existem diversas formas de distribuir os diferentes diretórios para um aplicação com o Zend Framework. Neste exemplo introdutório, para não nos estendermos muito, vamos mostrar apenas uma forma de organização que irá ajudar posteriormente no entendimento da forma de organização adotada pelo Septo Framework.

Precisamos de três pastas básicas para começar. A primeira delas é a pasta pública ou *Web Root*. O nome desta pasta pode variar bastante dependendo da configuração do servidor web. Os nomes mais comuns, no entanto, costumam ser: *public*, *www* ou *public_html*. Vamos adotar a nomenclatura *www* para a pasta pública neste exemplo.

Na grande maioria dos sistemas PHP todo o código fonte do site costuma ficar centralizado nesta pasta. No nosso caso, preferimos isolar nesta pasta apenas responsabilidade de carregar e chamar o Front-Controller do Zend Framework deixando em outros lugares, conforme veremos mais adiante, a parte principal da aplicação. Além disso, também é nesta pasta onde ficam todas as imagens, folhas de estilo e *scrips* javascripts, ou seja, os arquivos realmente públicos do site, aqueles que são acessados diretamente através do *browser*.

Isolar na pasta *www* a chamada ao Front-controller e os arquivos públicos é uma prática encorajada pelos desenvolvedores do Zend Framework pois colabora para a segurança evitando que arquivos cruciais contendo o código fonte principal do site possam ser acessados diretamente através do navegador web.

Outra pasta necessária é a pasta onde deve ser colocado o código do Zend Framework e de outras possíveis bibliotecas utilizada pela aplicação como o Doctrine, por exemplo. Não há um padrão de nomenclatura para esta pasta e optamos por chamá-la aqui de *lib*.

A última pasta é aquela onde, de fato, estará a aplicação que dá característica ao site. É onde o desenvolvedor dedicará a maior parte do tempo construindo seus *controllers* e suas *views*. Também não há um padrão restritivo de nomenclatura para esta pasta, embora ela costume ser chamada de *application* ou *app*. Optamos aqui por *app*.

Sendo assim temos aqui a descrição das três pastas essenciais para fazer rodar uma aplicação que utiliza do Zend Framework. Abaixo, apresentamos

a estrutura completa contendo inclusive as sub-pastas que compõem estas três pastas principais.


Nesse momento ao acessarmos o site nada irá acontecer. Isso porque não existe nada na pasta pública ainda. E o primeiro arquivo de devemos criar nesta pasta é o **.htaccess**. O conteúdo deste arquivo será algo semelhante ao exibido abaixo:

```
RewriteEngine on
RewriteRule !\.(js|ico|gif|jpg|png|css)$ index.php
```

Este arquivo está reescrevendo todas as URLs que não apontam para nenhum arquivo público como uma imagem, uma folha de estilo ou um arquivo *javascript*. O motivo de fazer isso é para permitir que tenhamos tipos muito mais amigáveis de URLs e que através dessas URLs amigáveis seja possível acionar a *action* de um determinado *controller*. Comparemos, por exemplo, as duas URLs abaixo:

```
http://www.septograma.com.br/index.php?controller=post&action=list&dia=1&mes=6&ano=2009
```

```
http://www.septograma.com.br/post/list/dia/22/mes/6/ano/2009
```

Ambas URLs possuem o mesmo propósito e devem retornar o mesmo tipo de resultado. Mesmo assim, elas são bastantes diferentes porque, no

primeiro caso, temos uma URL que não utiliza os recursos de reescrita de URL e, no segundo, podemos ver como a reescrita de URL pode contribuir para a construção de URLs mais simples, claras, amigáveis e fáceis de memorizar.

Nem todos os servidores vêm com este recurso de reescrita de URL ativado. No caso do Apache, por exemplo, é preciso que a cláusula *rewrite_mode* esteja ativada para que tudo funcione. Com isso resolvido, podemos dar início ao entendimento de como fazer o Zend Framework rodar utilizando os recursos MVC que possui.

### *3.3.2 Preparando o Arquivo de Bootstrap*

O segundo arquivo que deve conter na pasta *www* é o **index.php**. Este arquivo neste contexto é conhecido como arquivo de *bootstrap* uma vez que sua função é a de carregar todo o ambiente.

O conteúdo do arquivo de *bootstrap* pode variar bastante de uma implementação para outra. A intenção aqui é oferecer um exemplo esclarecedor contendo o básico necessário para rodar o MVC do Zend Framework.

Abaixo, o código de *bootstrap*:

```
01  <?php
02
03  error_reporting(E_ALL|E_STRICT);
04  ini_set("display_errors", "on");
05
06  set_include_path("../lib".PATH_SEPARATOR.get_include_path());
07
08  require_once "Zend/Loader/Autoloader.php";
09  Zend_Loader_Autoloader::getInstance()->setFallbackAutoloader(true);
10
11  $front = Zend_Controller_Front::getInstance();
12  $front->setControllerDirectory("../app/controllers");
13  $front->throwExceptions(true);
14  $front->dispatch();
```

Nas linhas 3 e 4 é dito que todo erro que ocorrer deverá ser exibido no *browser*. Configuramos isso dessa maneira agora por considerar que no momento o site está sendo desenvolvido, mas é importante que tais erros não sejam exibidos quando chegar o momento da publicação do site. Para isso, bastará trocar o 'on' da linha 4 por 'off'.

A linha 6 coloca o Zend Framework e tudo que estiver na pasta lib no path e dessa forma é possível usar o Zend Framework e demais componentes como se estivesse no contexto de qualquer *script* php dentro do site.

A linha 8 e a linha 9 ativam um recurso bastante interessante do Zend que reflete no carregamento de classes em toda a aplicação. Na linha 8 estamos dizendo para carregar a classe **Zend_Loader_Autoloader**. Perceba que o caminho completo para o arquivo que contém esta classe na verdade é ***'/lib/Zend/Loader/Autoloader.php'***, mas como na linha 6 a pasta lib foi inserida no path é possível carregá-lo apenas com o caminho ***'Zend/Loader/Autoloader.php'***.

A classe **Zend_Loader_Autoloader** implementa *singleton*, por isso, para conseguir uma instância desta classe, pedimos isso para o método estático e público **'getInstance'**. Logo após, dizemos para este objeto ativar o função de auto carregamento de classes implementado pelo Zend Framework.

Com este recurso ativado ficamos livre da maior parte do *includes* e *requires* tão comuns em PHP que infernizam a vida dos desenvolvedores. Para isto funcionar, basta passar a utilizar um padrão de nomenclatura bastante simples e intuitivo onde o próprio nome da classe explica como chegar ao arquivo. Isso evita que o programador tenha que informar explicitamente através de funções PHP como *require*, *include* ou *include_once* para carregar uma determinada classe antes que ela seja utilizada.

Na linha 11 devemos conseguir uma instância de **Zend_Controller_Front** para dar continuidade ao processo de inicialização de todo o ambiente. Fazemos isso utilizando o método **getInstance** uma vez que **Zend_Controller_Front** também implementa *singleton*.

Observamos aqui que graças à ativação da função de auto-carregamento do Zend Framework na linha 9 não foi necessário solicitar que a classe **Zend_Controller_Front** fosse carregada antes de utilizá-la, uma vez que, pelo próprio nome da classe, o ambiente já sabe que o caminho para o arquivo é ***'Zend/Controller/Front.php'*** tendo condições de fazer o carregamento 'por debaixo dos panos'.

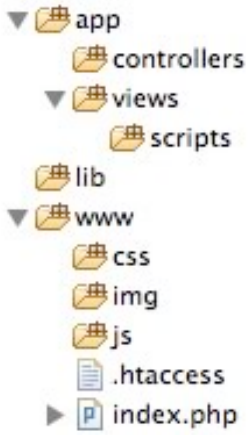
Na linha 12 é informado para o *Front-controller* onde estão os arquivos *controllers* da aplicação. Estes arquivos são fundamentais pois com

base na *URL* informada o *Front-controller* irá direcionar a solicitação para alguns arquivos desta pasta.

Na linha 13 é informado para o *Front-controller* exibir as exceções que surgirem no *browser*. Vamos ativar esta opção pois, deste ponto em diante, continuaremos construindo a aplicação com base nos erros lançados pelo *Front-controller*.

### *3.3.3 Construindo Controllers, Actions e Views*

Com tudo que foi feito até aqui a aparência da estrutura de diretórios criadas deverá estar de acordo com a imagem abaixo:




Agora, graças ao arquivo de *bootstrap* e a reescrita de *URL*, o **Zend_Controller_Front** já pode assumir a responsabilidade por fazer o lançamento da aplicação e procurar pela *action* que deve atender a uma determinada solicitação. Mas o que é uma *action* para o Zend Framework e como chamá-la através de uma *URL*?

Basicamente, uma *action* é um método público dentro de uma classe que é um *controller*. Se pedirmos para rodar a aplicação da forma que está, sem ter construído *controller* algum, vamos receber o seguinte erro:

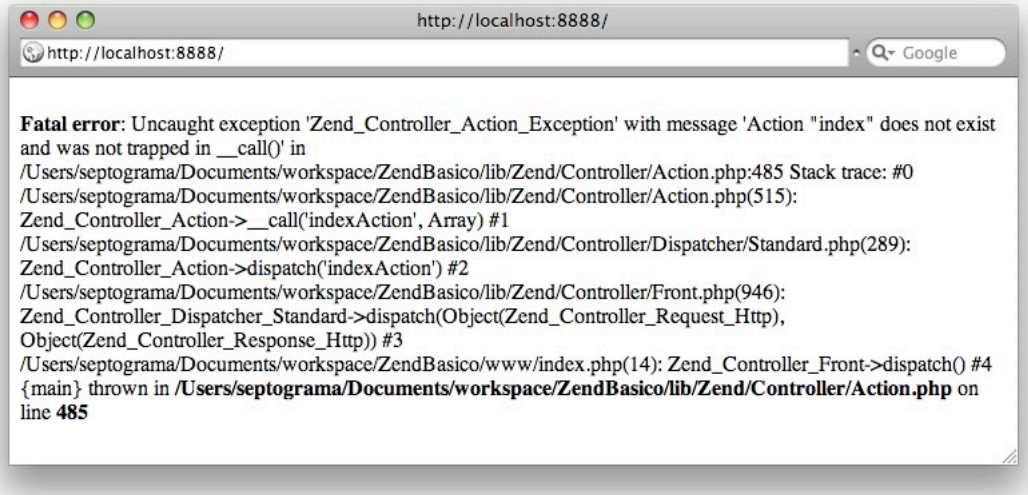

Como se pode observar, uma exceção do tipo **Zend_Controller_Dispatcher_Exception** é lançada e é informado que um controle chamado é inválido. A mensagem também diz que o *controller* chamado foi o *controller index*, embora na URL não tenhamos informado isso explicitamente. A seguir, explicaremos como isso funciona. Por enquanto, vamos acreditar na mensagem e construir um *controller* chamado *IndexController*. No diretório onde ficam os *controllers* deve ser criado um arquivo chamado **IndexController.php**:

```
01  <?php
02
03  class IndexController extends Zend_Controller_Action {
04
05  }
```

Este arquivo, para ser considerado um *controller* pelo Zend Framework precisa atender a quatro critérios, são eles:

1. Estar numa pasta onde ficam os controllers;
2. Ter um nome de arquivo que termina com "**Controller.php**", no caso, o arquivo do controller index chama-se **IndexController.php**;
3. Ter um nome de classe que termina com "**Controller**", no caso, o *controller index* terá o nome **IndexController**.
4. Ser subclasse de **Zend_Controller_Action**.

Seguido estes critérios, temos o nosso *controller*. Vamos ver o que acontece agora ao tentar rodar a aplicação novamente:



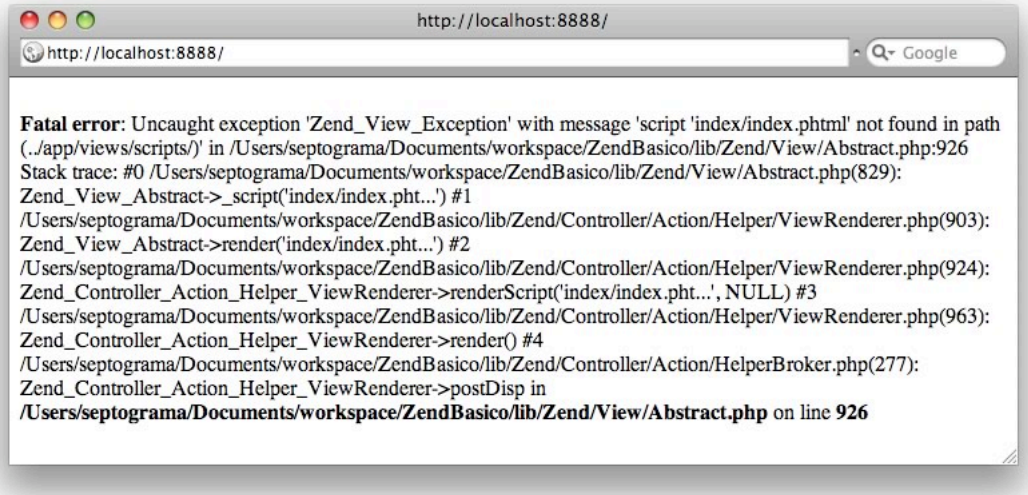

Desta vez, uma **Zend_Controller_Action_Exception** é disparada informando que não encontrou uma *action* chamada **index**. Observamos novamente que na *URL* não informamos nada explicitamente para a *action* **index**. Por enquanto, vamos aceitar essa informação e, a seguir, explicar como diferentes *controllers* e *actions* são chamados através das *URLs*.

Para criar a *action* **index**, basta abrir o nosso arquivo de controller recém criado e inserir um método público chamado **indexAction** dentro:

```
01  <?php
02
03  class IndexController extends Zend_Controller_Action {
04
05    public function indexAction() {
06
07    }
08
09  }
```

As *actions* também seguem um padrão de nomenclatura e seu nome sempre deve ser todo em minúsculo seguido da terminação **"Action"** com o **"A"** maiúsculo. Como a intenção era criar a *action* **index**, o nome acabou sendo **indexAction**.

Temos então um *controller* e uma *action* definidos. Vejamos agora o que está faltando observando o próxima exceção lançada pelo *Front-controller*:




Para que tudo passe a funcionar, falta resolver está última exceção, uma **Zend_View_Exception**. Esta mensagem refere-se à um problema na **view**, ou seja, na parte do MVC responsável por criar o conteúdo que será será enviado e exibido pelo *browser*. Como estamos utilizando o padrão de projeto MVC, nada mais justo do que esperar que o conteúdo a ser exibido não se misture com a parte responsável por controlar as solicitação recebidas, ou seja, o código dos *controllers*.

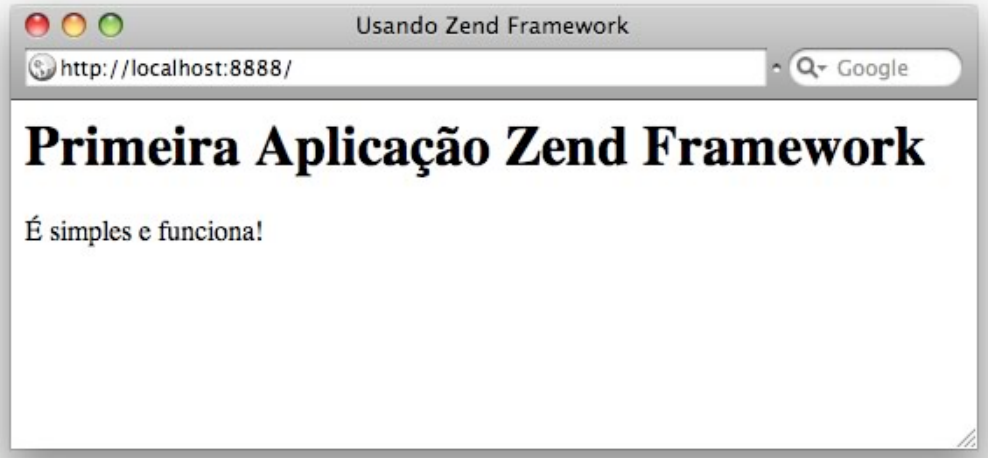
Portanto, o arquivo de *script* que está faltando é um arquivo que deverá ser colocado na *view*. No Zend Framework cada controller possui uma respectiva pasta dentro do diretório ***view*** e dentro desta pasta do *controller* na *view* existe um arquivo de *script* para cada *action* do *controller*.

A primeira vista, tudo pode parecer um pouco confuso e contra-produtivo, mas tão logo venha o entendimento do porquê disso, logo percebemos que, por detrás desta aparecente confusão, está um grande auxílio oferecido pelo Zend Framework para produzir sistemas *web* mais organizados e fáceis de manter fugindo do famoso "código-espaguetti".

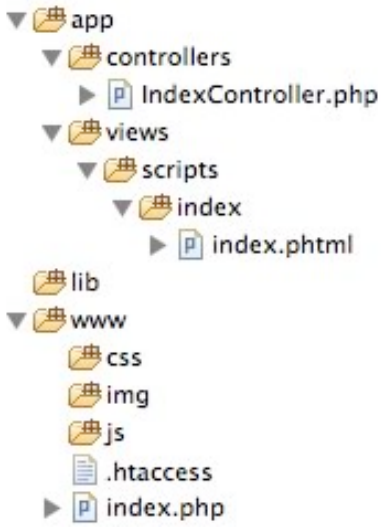
Para criar o arquivo da *view* que está faltando vamos até a pasta *views* e criamos lá dentro a pasta chamada ***scripts*** e dentro da pasta ***scripts*** vamos criar outra chamada ***index***. Com esta ação é como se estivessemos dizendo para a aplicação que vamos colocar ali, nesta pasta ***index***, todos os arquivos de *script* relacionados ao **IndexController**. Criaremos agora o arquivo **index.phtml** dentro da pasta ***index***:

```
01    <html>
02          <head>
03                <title>Usando Zend Framework</title>
04          </head>
05          <body>
06                <h1>Primeira Aplicação Zend Framework</h1>
07                <p>É simples e funciona!</p>
08          </body>
09    </html>
```

Finalmente, o resultado exibido pelo *browser* será:




Ao final deste exemplo inicial, a estrutura de diretórios de todo o sistema ficou da seguinte forma:



## 3.4 Criando Novas Actions

Agora que a primeira *action* foi criada, torna-se simples expandir a aplicação através da criação de novos *controllers*, *actions* e *scripts*. Vamos criar portanto novas páginas para esta primeiro *controller* com o objetivo de reforçar a compreensão da dinâmica que existe em *controllers*, *actions*, *scripts* e *URLs* amigáveis no Zend Framework.

Na sessão anterior, para exibir a primeira página da nossa aplicação foi necessário criar um *controller* **index** e uma *action* **index**. A *URL* para acessar este recurso, a rigor, seria algo do tipo:

http://localhost/index/index

Na URL acima, o primeiro *index* (esquerda para direita) refere-se ao *controller* que estamos chamando e o segundo *index* à action deste controller que está sendo disparada.

Mas se prestarmos atenção, tudo que tivemos de digitar para ativar a *action* **index** do *controller* **index**, foi:

http://localhost

Isso é possível pois, uma vez não especificado o controller e/ou *action*, o Zend entende que o usuário está buscando pelo *controller* e pela *action default*, ou seja, o *controller* e a *action* **index**.

Vamos agora criar uma nova *action* no **IndexController** para ver como deverá ficar a solicitação:

```
01  <?php
02
03  class IndexController extends Zend_Controller_Action {
04
05    public function indexAction() {
06
07    }
08
09    public function outraAction() {
```

```
10
11      }
12
13  }
```

Inserida a nova *action* com o nome de “outra”, nas linhas 9 à 11, vamos criar agora um novo arquivo de *script* – **outra.phtml** – para esta *action* na *view*. Assim, evitamos receber um **Zend_View_Exception**:

```
01  <html>
02      <head>
03          <title>Usando Zend Framework</title>
04      </head>
05      <body>
06          <h1>Primeira Aplicação Zend Framework</h1>
07          <p>Está é a página da outra action.</p>
08      </body>
09  </html>
```

Para acessar este novo recurso pelo *browser* teremos que informar o seguinte endereço:

http://localhost/index/outra

E o resultado será:

Observe que, desta vez, não há outra opção senão informar ao navegador qual o *controller* (**index**) e qual é a *action* (**outra**), isso porque já não estamos mais acessando a *action* **index** do *controller* **index** que são *default*.

## 3.5 Criando Novos *Controllers*

Entendido o processo de criação de novas *actions*, cabe agora mostrar como funciona a criação de novos *controllers* e como isso irá refletir nas *URLs* e nos arquivos da *view*.

Na mesma pasta onde se encontra o arquivo **IndexController.php** criaremos um novo arquivo de *controller* chamado **NovoController.php**. O *controller* **novo** terá duas *actions, são elas*: **index** (a sua *action default*) e **contato**, ficando da seguinte forma:

```
01  <?php
02
03  class NovoController extends Zend_Controller_Action {
04
05    public function indexAction() {
06
07    }
08
09    public function contatoAction() {
10
11    }
12
13  }
```

Agora que temos o arquivo do *controller* com as suas respectivas *actions* criadas, falta resolver os aspectos relacionados à camada de apresentação, ou seja, temos ainda que criar os arquivos de *script* para cada uma das novas *actions* deste *controller*.

Como se trata de um novo controller, não podemos simplesmente colocar os novos *scripts* dentro da pasta ***/app/views/scripts/index***, onde vinhamos colocando os scripts relacionados ao *controller* **index**. Para os *scripts* deste novo controller teremos uma nova pasta chamada ***novo*** dentro da pasta de *scripts*. E vamos criar em ***/app/views/scripts/novo*** um arquivo para cada umas das *actions* de **NovoController**.

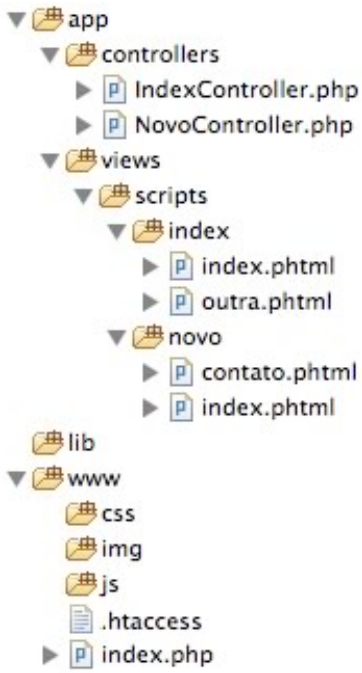
O arquivo da *view* para a *action* **index** de **NovoController** deverá chamar-se **index.phtml** e poderá ser assim:

```
01    <html>
02          <head>
03                <title>Usando Zend Framework</title>
04          </head>
05          <body>
06                <h1>Primeira Aplicação Zend Framework</h1>
07                <p>Estou no controller <b>novo</b> e na action
      <b>index</b>.</p>
08          </body>
09    </html>
```

E o arquivo da *view* para a *action* **contato** do Novo Controller deverá se chamar **contato.phtml** poderá ser assim:

```
01    <html>
02          <head>
03                <title>Usando Zend Framework</title>
04          </head>
05          <body>
06                <h1>Primeira Aplicação Zend Framework</h1>
07                <p>Escreva para nós: nos@empresa.com.br</p>
08          </body>
09    </html>
```

Com essas alterações, nossa aplicação deverá ficar da seguinte forma:




Para acessar a *action* **index** de **NovoController** a *URL* deverá ser:

http://localhost/novo/index

Ou ainda, se preferir, a URL abaixo também leva para o mesmo local uma vez que, na ausência da especificação da uma *action*, o *Front-controller* chamará a *action default*, ou seja, a **index**.

http://localhost/novo

E, para acessar a *action* contato de **NovoController**, o caminho será:

http://localhost/novo/contato

Neste momento, já deve estar bem claro o entendimento da dinâmica existente no Zend Framework entre *controllers*, *actions*, *views* e *URLs* amigáveis. Logo adiante, apresentaremos um novo conceito, o de módulos.

Uma vez entendido o que foi apresentado até aqui e somando-se a isso o conceito de módulos, que ainda nos falta abordar, teremos as noções

necessárias para entender como o Zend Framework foi incorporado ao Septo Framework. Mas antes de falar de módulos, vamos conhecer um outro recurso do Zend, o **Zend_Layout**.

## 3.6 Ativando o Recurso de Layout

O entendimento dos recursos oferecidos pela classe **Zend_Layout** do Zend Framework não é fundamental para que se possa compreender o Septo Framework de maneira superficial. Mas, por se tratar de um recurso bastante interessante e fácil de dominar, preferimos incluí-lo neste trabalho.

Nos exemplos de arquivos que compuseram a *view* apresentados até aqui é possível observar que toda a estrutura HTML é basicamente a mesma e tudo que mudou de um *script* para outro foi apenas uma linha, a linha 7. Com o **Zend_Layout** podemos separar facilmente aquilo que muda daquilo que é o esqueleto do site, o *layout*.

Para ativar o recurso vamos inserir apenas um nova linha no arquivo de *bootstrap*, o **index.php** na pasta pública, conforme mostrado abaixo:

```
01  <?
02
03  error_reporting(E_ALL|E_STRICT);
04  ini_set("display_errors", "on");
05
06  set_include_path("../lib".PATH_SEPARATOR.get_include_path());
07
08  require_once "Zend/Loader/Autoloader.php";
09  Zend_Loader_Autoloader::getInstance()->setFallbackAutoloader(true);
10
11  Zend_Layout::startMvc("../app/views/layouts/");
12
13  $front = Zend_Controller_Front::getInstance();
14  $front->setControllerDirectory("../app/controllers");
15  $front->throwExceptions(true);
16  $front->dispatch();
```

A única diferença em relação ao arquivo de *bootstrap* anterior está na linha 11. Nesta linha chamamos o método estático **startMvc** da classe **Zend_Layout** passando como parâmetro o local onde colocaremos os arquivos de *layout*.

Observe que ainda não temos este diretório em nossa aplicação e muito menos o arquivo de *layout* que deve ficar dentro dele. Para evitar ficar

recebendo exceções, vamos logo criar o arquivo padrão de *layout*. Um arquivo padrão de *layout* é aquele que será utilizado para todas as chamadas, a não ser que seja informado nos *controllers* que é para ser cagerrado um *layout* diferente do *default*.

Para o Zend Framework, o arquivo de *layout default* chama-se **layout.phtml**. E tudo que temos que fazer agora é criar um arquivo com este nome e salvá-lo na nova pasta ***/app/views/layouts***.

Este arquivo, terá o seguinte conteúdo:

```
01  <html>
02    <head>
03      <title>Usando Zend Framework</title>
04    </head>
05    <body>
06      <h1>Primeira Aplicação Zend Framework</h1>
07      <?= $this->layout()->content ?>
08    </body>
09  </html>
```

O que isso irá fazer basicamente é repetir para qualquer solicitação as linhas que vão de 1 até 6 e de 8 até 9 e será "colado" na linha 7 somente o conteúdo de *script* específico da *action* chamada.

Feito isso, o recurso de *layout* já estará funcionando. Só teremos agora de lembrar de editar todos os arquivos de *scripts* criados até aqui e manter neles apenas o conteúdo que estava originalmente na linha 7.

Com o **Zend_Layout** em ação, além de facilitar a criação de novas telas para a aplicação evitando redigitar ou ficar copiando código, fica muito mais fácil e produtivo manter as telas existentes uma vez que as alterações em apenas um arquivo, o arquivo de *layout*, será refletida em todas as demais telas da aplicação.

## 3.7 Trabalhando com múltiplos *modules*

### *3.7.1 Visão Geral*

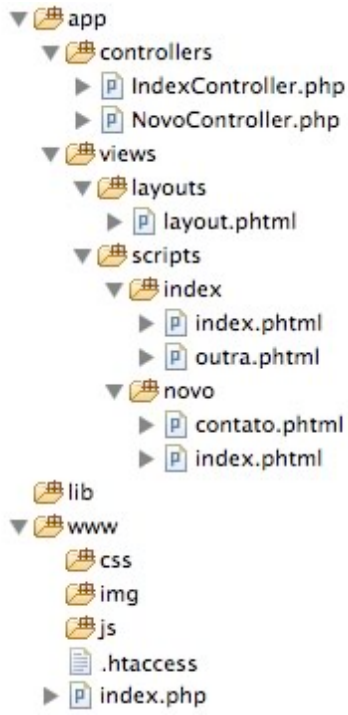
A última coisa que temos que saber a respeito do Zend Framework antes de partimos para o Septo Framework é a maneira como ele é capaz de trabalhar utilizando mútiplos *modules*.

Trabalhar com múltiplos *modules* significa ter um conjunto de pastas contendo seus *controllers*, suas *views* e outros arquivos e pastas específicos. A principal vantagem de ativar o recurso de módulos do Zend Framework é que isso permite uma melhor organização dos recursos dentro da aplicação.

Para o Zend Framework a utilização de mútiplos *modules* é opcional, mas para o Septo Framework funcionar ela é obrigatória, uma vez que foi ativando esta funcionalidade que encontramos a melhor solução para distribuir os Septo Modules, conforme veremos mais adiante.

### *3.7.2 Reorganizando os diretórios*

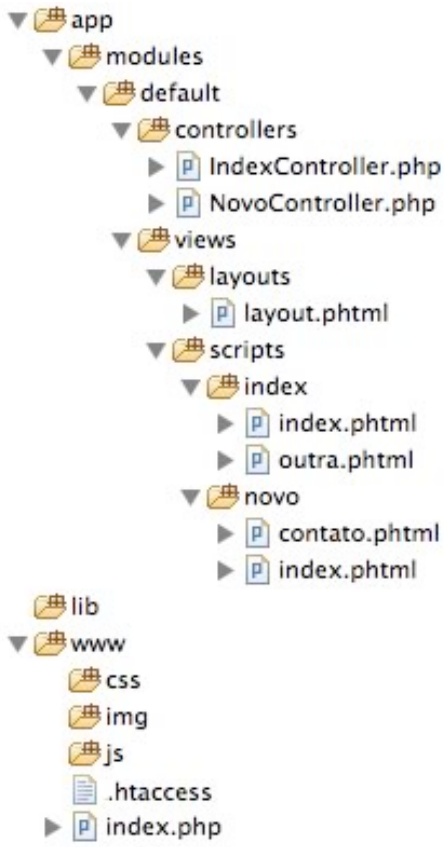
Até o momento, a estrutura de diretórios de toda aplicação até este ponto deve estar da seguinte forma:




Como nossa intenção é passar a utilizar o recurso de mútiplos *modules* precisamos modificar esta estrutura para atender a nova exigência. Fazemos isso criando uma pasta chamada *modules* dentro de *app* e criando dentro da pasta *modules* os respectivos módulos da aplicação.

Num primeiro momento, o único módulo que teremos será o módulo padrão cujo nome é *default*. E levaremos para dentro da pasta *default* as

pastas *controllers* e *views* que antes estavam diretamente em *app*. No final destas alterações a aplicação passará a ter o seguinte aspecto:




### *3.7.3 Adaptando o arquivo de bootstrap*

Uma vez adaptada a estrutura de diretórios, a última coisa que temos que fazer é modificar o arquivo de bootstrap. A principal alteração que iremos fazer será substituir a linha 14 onde dizia:

```
14  $front->setControllerDirectory("../app/controllers");
```

Por esta linha:

```
14  $front->addModuleDirectory("../app/modules/");
```

Ou seja, deixamos de informar para o **Front-controller** onde estão os controllers da aplicação e passamos a informar onde estão os módulos da aplicação.

A seguinte dúvida pode surgir nesse momento: se deixamos de informar onde estão os *controllers*, como que o **Front-controller** continua sabendo onde eles estão?

O **Front-controller** continua sabendo onde estão os *controllers* pois parte do pressuposto que dentro da pasta de cada um dos *modules* irá encontrar uma pasta chamada *controllers* onde encontram-se os *controllers*.

A mesma lógica funcionou deste o início para a pasta *view*, uma vez que nunca foi necessário informar onde estão os arquivos da *view*. Neste caso, o **Front-controller** simplesmente partiu do pressuposto de que a pasta *views* deveria estar no mesmo local onde se encontra-se a pasta dos *controllers*.

Essa alteração no arquivo de *bootstrap* já seria necessária para ativar o recurso de múltiplos módulos do Zend Framework. Porém, na linha 11 do nosso arquivo de *bootstrap*, havíamos ativado o recurso de *layout* e informado que os arquivos de *layout* se encontravam-se na pasta ***/app/views/layouts***. Como isso acabou mudando depois de reorganização que fizemos nos diretórios para trabalhar com múltiplos módulos, ainda temos que fazer uma pequena adaptação nessa linha antes de ver este recurso funcionar. No final, o arquivo de *bootstrap* completo deverá ficar da seguinte forma:

```
01  <?
02
03  error_reporting(E_ALL|E_STRICT);
04  ini_set("display_errors", "on");
05
06  set_include_path("../lib".PATH_SEPARATOR.get_include_path());
07
08  require_once "Zend/Loader/Autoloader.php";
09  Zend_Loader_Autoloader::getInstance()->setFallbackAutoloader(true);
10
11  Zend_Layout::startMvc("../app/modules/default/views/layouts/");
12
13  $front = Zend_Controller_Front::getInstance();
14  $front->addModuleDirectory("../app/modules/");
15  $front->throwExceptions(true);
16  $front->dispatch();
```

### *3.7.4 O que muda nas URLs*

Anteriormente vimos que para acessar uma determinada *action* de um *controller* bastava digitar algo do tipo:


Agora, além do nome do *controller* e da *action*, informamos também o nome do *module* que está sendo utilizado:


É claro que o exemplo acima trata de chamar o *module*, o *controller* e a *action default*. Isso significa que apenas digitando localhost a mesma action seria ativada. O importante é compreender que, para qualquer situação, está é uma forma que sempre irá funcionar.

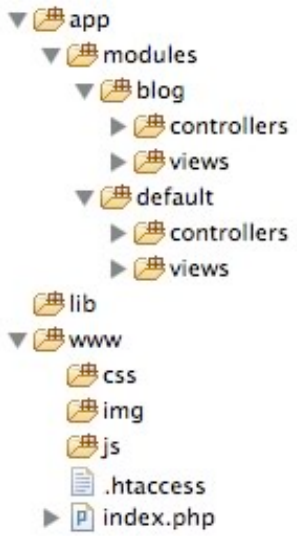
### *3.7.5 Criando um novo módulo*

Pequenos problemas podem se converter facilmente num grande motivo para abandonarmos uma boa ferramenta. E existe algo relacionado a criação e utilização de novos *modules* no Zend Framework que nos atrapalhou bastante no início e não foi fácil achar um material que fosse capaz de prever e sanar nossa dúvida.

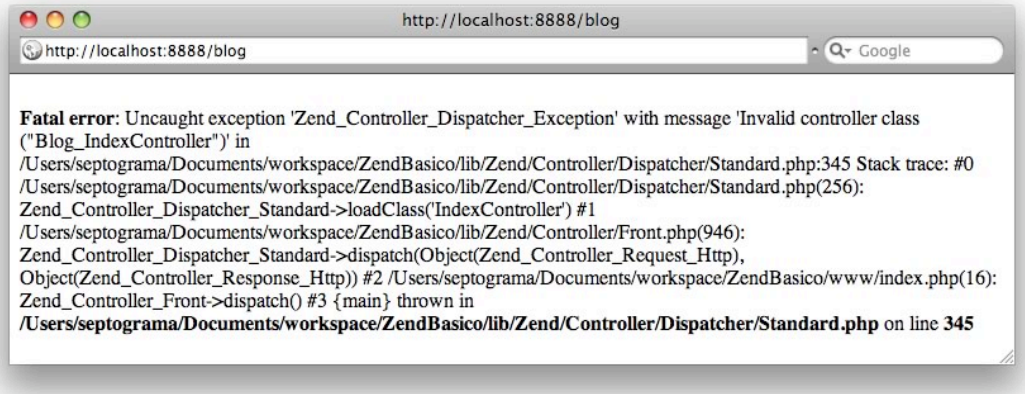
Por isso, embora já tenhamos falado daquilo que é mais relevante para passar a utilizar o recurso de múltiplos módulos do Zend Framework, vamos dedicar um pouco mais de atenção ao assunto para cobrir um detalhe que pode atrapalhar a vida dos desenvolvedores quando se encerra a teoria e começa a parte prática.

Para isso, criaremos um novo módulo que vamos chamar de *blog*. O objetivo aqui será apenas criar o módulo e, em seguida, definir um *controller*, uma *action* e uma *view* para ele. Depois vamos testar algumas *URLs* e mostrar aquilo que nos confundiu quando, pela primeira vez, utilizamos o recurso de múltiplos módulos do Zend Framework.

Podemos fazer isso rapidamente fazendo uma cópia da pasta **default**, *module* que já criamos, na pasta modules e renomeá-la para "blog". Agora, a estrutura da aplicação deverá estar com a seguinte forma:




Como se pode ver, temos agora dois *modules* e, aparentemente, tudo já devia estar funcionando. Mas, ao tentamos acessar um *controller* ou uma *action* no novo *module* criado, recebemos a seguinte exceção:




Trata-se de um erro simples e fácil de corrigir, mas custamos um pouco até entendermos o que a exceção estava tentando dizer. E o que ela está tentando dizer é que o padrão de nomenclatura da classe dos *controllers* é diferente quando não estamos em um *module* que não seja o *default*.

Neste caso, estamos tentando acessar um *controller* do módulo *blog* e, por ser um módulo diferente do *default*, toda classe de *controller* deste módulo deverá conter na frente do seu nome o prefixo "Blog_". O **IndexController** do *module* **Blog**, por exemplo, deverá se chamar **Blog_IndexController**.

Essa alteração deve ser feita apenas no nome da classe do *controller*, é importante lembrar que, embora o nome da classe mude, o nome do arquivo continua igual, ou seja, **IndexController.php**.

# 4. DOCTRINE

## 4.1 Visão Geral

O Dotrine é uma ferramenta ORM para PHP5 e simplifica a forma de tratar com entidades persistíveis. É uma ferramenta com inúmeros recursos relacionados a ORM e é inspirada no Hibernate, escrito na linguagem Java.

Para fins de esclarecer a participação desta ferramenta no Septo Framework, vamos apresentá-la focando naquilo que é importante no sentido de apoiar o entendimento deste trabalho.

Para isso, iremos criar uma nova aplicação de exemplo apenas para testar o Doctrine com uma estrutura de diretórios muito semelhante a apresentada no capítulo anterior, onde falamos sobre o Zend Framework.

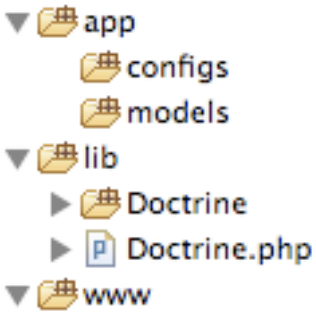
## 4.2 Instalando o Doctrine

O primeiro passo será baixar o Doctrine no site *doctrine-project.org*, descompactá-lo e colocá-lo na sua devida pasta, no nosso caso, a mesma pasta ***lib*** onde havíamos colocado o Zend Framework.

Na pasta ***app***, criaremos mais outras duas. Uma será ***models*** e outra será ***configs***. Na primeira, estarão as nossas entidades em forma de classe; na segunda, nossas entidades descritas em arquivo YAML.

Para carregar o ambiente Doctrine, seguiremos a mesma lógica do carregamento do ambiente Zend Framework, construindo um arquivo de *bootstrap* dentro da pasta pública, a ***www***.

Com tudo isso feito, a aparência geral da estrutura de diretórios de toda a aplicação deverá ser a seguinte:




O próximo passo será preencher agora o arquivo de *bootstrap* para fazer o Doctrine rodar. Embora estejamos seguindo a mesma estrutura básica de diretórios utilizada no capítulo anterior, não vamos colocar nada no arquivo de *bootstrap* relacionado ao carregamento do Zend Framework uma vez que pretendemos apresentar o Doctrine de maneira isolada para facilitar o entendimento de cada uma das ferramentas separadamente primeiro. Mais adiante, será mostrado esta integração acontecendo no Septo Framework.

Cientes disso, o arquivo de *bootstrap* para carregar o Doctrine irá se chamar **bootstrap.php**, ficará na pasta pública, *www*, e terá a seguinte configuração:

```
01  <?
02
03  error_reporting(E_ALL|E_STRICT);
04  ini_set("display_errors", "on");
05
06  require_once(dirname(__FILE__) . '/../lib/Doctrine.php');
07
08  spl_autoload_register(array('Doctrine', 'autoload'));
09
10  $dsn = 'mysql:dbname=db_doctrine;host=localhost';
11  $user = 'root';
12  $password = 'root';
13
14  $dbh = new PDO($dsn, $user, $password);
15  $conn = Doctrine_Manager::connection($dbh);
```

Na linha 3 e 4 estamos informando para à aplicação que exiba todos os possíveis erros ou alertas que surgirem durante a execução.

A linha 6 carrega a classe Doctrine que foi colocada dentro da pasta ***lib***.

A linha 8 ativa o recurso *autoload* do Doctrine. No capítulo anterior mostramos o **Zend_Autoload** sendo carregado e foi explicado o impacto disso na

aplicação. Tanto o *Autoload* do Doctrine quando o *Autoload* do Zend tem o mesmo propósito e fazem exatamente a mesma coisa. É desaconselhável carregar ambos recursos de *Autoload* ao mesmo tempo, haja vista que apenas um deles é o suficiente para ativar o recuso em toda a aplicação.

No Septo Framework, onde Doctrine e Zend são utilizados conjuntamente, optamos por ativar apenas o *autoload* do Zend Framework. Neste exemplo, por estarmos utilizando o Doctrine isoladamente, ativamos seu recurso de *autoload* na linha 8.

Nas linhas 10, 11 e 12 são criadas três variáveis que servirão de parâmetro para o método construtor de PDO utilizado na linha 14. Na linha 10 informamos que o tipo de base de dados utilizada é *mysql*, que o nome do banco de exemplo é *db_doctrine* e que a localização da máquina contendo o banco na rede é *localhost*. Na linha 11 e 12 informamos o nome do usuário e a sua senha de acesso ao banco de dados, respectivamente.

Com estas variáveis das linha 10, 11 e 12, criamos o objeto PDO na linha 14. O PDO é uma extensão na linguagem PHP que define uma interface leve e consistente voltada para o acesso ao banco de dados em PHP[5].

Na linha 15 é aberta uma conexão com o a base de dados intermediada pelo **Doctrine_Manager** passando o objeto PDO como parâmetro para o método estático *connection*. A partir desse ponto, já é possível utilizar os recursos do Doctrine em nossa aplicação.

## 4.3 Gerandos arquivos e tabelas com o Doctrine

### *4.3.1 Visão Geral*

Chegou o momento de mostrar o Doctrine em ação e, levando em consideração que o Septo Framework é voltado para trabalhar com entidades e que foi somente através do Doctrine e graças a sua valiosa contribuição que tornou-se possível concluir este projeto, iremos procurar apresentar da maneira significativa e em detalhes o porquê de ficarmos tão entusiasmados com o potencial desta ferramenta através de exemplos daquilo que o Doctrine é capaz de fazer.

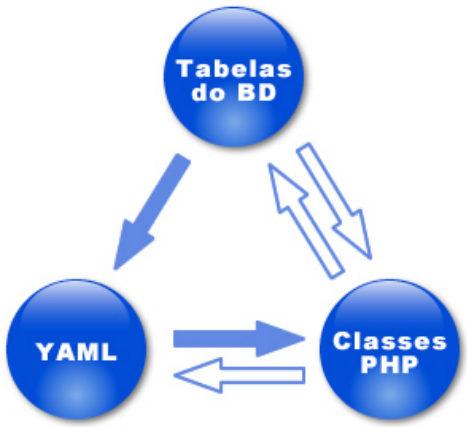
---

[5] http://br.php.net/pdo

Para começar, com o Doctrine é possível gerar os três ingredientes fundamentais para a criação de aplicações utilizados pelo Septo Framework, são eles:

1. Arquivo de configuração YAML;
2. Classes PHP do modelo;
3. Tabelas na base de dados.

Para isso, precisamos partir de algum ponto para começar. Temos que escolher se começaremos criando o arquivo de configuração YAML, as classes do modelo ou as tabelas no banco de dados. Uma vez definido isso, as possíveis transformações realizadas pelo Doctrine são mostradas na figura abaixo:




As setas com a cor de preenchimento sólida indicam um caminho que nós encorajamos que seja seguido. As setas com as cores vazadas são transformações que o Doctrine também é capaz de fazer mas que, conforme mostraremos mais adiante, evitá-las pode ser uma boa prática.

Portanto, o caminho proposto por nós é criar primeiro as tabelas no banco de dados, depois o arquivo YAML e, por último, as classes PHP do modelo.

Muito embora este seja o caminho proposto, neste primeiro momento, faremos diferente: a primeira coisa que criaremos será o arquivo YAML, a segunda serão as classes PHP e, por último, criaremos as tabelas no banco.

### *4.3.2 Construindo o arquivo YAML*

Yaml é um padrão de serialização de dados humanamente amigável para todas as linguagens de programação[6] e foi o padrão escolhido pelo Doctrine para descrever as entidades.

Nesta primeira aplicação utilizando o Doctrine vamos escrever um arquivo YAML bastante simples cujo objetivo é descrever um cliente. Este cliente possui um id, um nome e um email.

```
01  Cliente:
02    tableName: clientes
03    columns:
04      id:
05        type: integer(8)
06        primary: true
07        autoincrement: true
08      nome:
09        type: string(255)
10      email:
11        type: string(255)
```

Vamos salvar este arquivo com o nome **entities.yml** e colocá-lo dentro da pasta ***configs***. O entendimento do arquivo acima é bastante intuitivo, mesmo assim, iremos explicá-lo.

Na linha 1 é informado qual será o nome da classe PHP que irá representar a entidade. Na linha 2, indentada com dois espaços, seguindo o padrão yaml, refere-se a uma propriedade deste cliente que estamos descrevendo, esta propriedade é a *tablename* que indica como deverá se chamar a tabela deste cliente no banco de dados. Resumindo, a linha 1 informa o nome da classe PHP e a linha 2 o nome da tabela.

Na linha 3 informamos que vamos começar a mapear os atributos do cliente para as colunas na tabela do banco. O primeiro atributo mapeado foi o id, com sua definição iniciando na linha 4 e terminando na linha 7. De acordo com a descrição, o id é um atributo do tipo inteiro (linha 5), é a chave-primária desta entidade (linha 6) e é auto-incremental (linha 7).

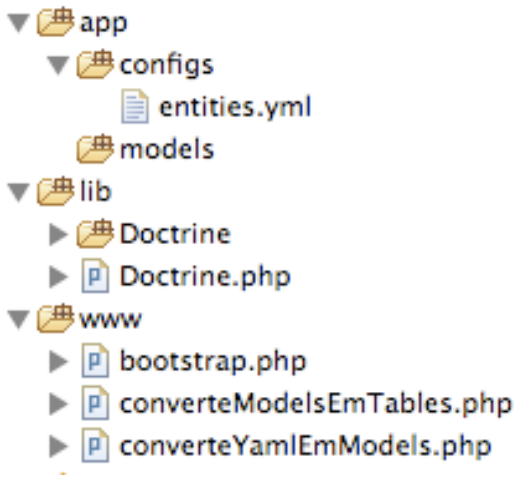
---

[6] http://www.yaml.org/

Nas linhas seguintes (8 a 11) mapeamos os atributos nome e email, ambos do tipo *string* com o tamanho de 255 caracteres.

Construído o arquivo YAML que mapeia as entidades, vamos criar agora dois arquivos na pasta ***www***: **converteYamlEmModels.php** e **converteModelsEmTables.php**. No primeiro, mostraremos como gerar as classes PHP do modelo partindo do arquivo de configuração YAML; no segundo, como gerar a tabela no banco de dados apartir das classes PHP do modelo.

Toda a estrutura da aplicação até este ponto deve estar conforme mostrado abaixo:



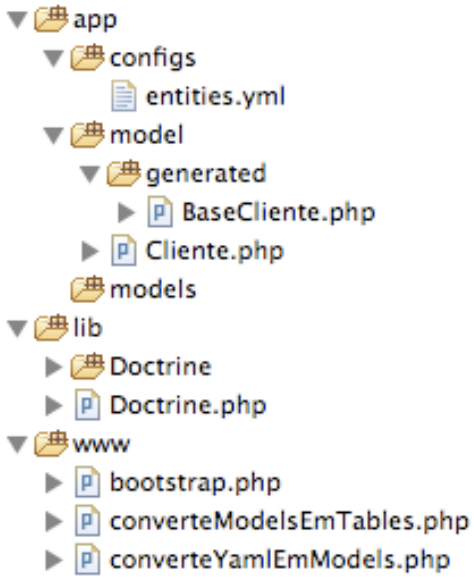

### *4.3.3 Criando as classes do modelo a partir do YAML*

O Arquivo **converteYamlEmModels.php**, cuja função é testar a geração das classes PHP do modelo a partir do arquivo de configuração YAML, deve ficar conforme mostrado a seguir:

```
01  <?php
02
03  require_once("bootstrap.php");
04
05  Doctrine::generateModelsFromYaml('../app/configs/entities.yml',
    '../app/model');
06
07  echo "As classes PHP foram geradas.";
```

Na linha 3 utilizamos o arquivo de *bootstrap* para carregar o ambiente do Doctrine. Na linha 5 chamamos o método estático da classe Doctrine que cria as classes do modelo passando dois parâmetros para ele: o caminho para chegar até o arquivo YAML e o local onde deverão ser colocas as classes que estão sendo criadas. A linha 7 informa apenas que o procedimento terminou.

Abrimos agora o navegador e pedimos para executar as instruções deste arquivo. Com esta operação realizada, a aplicação deverá ficar com o seguinte formato:




Conforme podemos observar, o Doctrine criou dentro da pasta ***app*** a pasta ***models*** onde está a nossa classe **Cliente**, fazendo parte do modelo. Observamos também que dentro da pasta ***models*** foi criada uma outra pasta chamada ***generated***, onde encontra-se uma outra classe chamada **BaseCliente** e tudo aquilo que foi escrito no arquivo YAML referente a entidade cliente está refletido aqui na forma de código PHP.

A classe **Cliente** estende **BaseCliente** que, por sua vez, estende **Doctrine_Record**. Mais adiante, falaremos das implicações que estas heranças múltiplas trouxeram para o nosso projeto. Por enquanto, é importante saber que o mapeamento que importa para o Doctrine é aquele que está escrito na classe em formato de código PHP e não o do arquivo YAML. Para o Doctrine, o arquivo

YAML é só uma forma mais fácil de conseguir gerar as classes bases, como **ClienteBase**, por exemplo.

### *4.3.4 Criando as Tabelas da Base de Dados a partir das Classes do Modelo*

Agora vamos fazer a última operação necessária para poder dar inícios aos primeiros testes de persistência das entidades através do Doctrine.

Já temos o arquivo de configuração YAML que nos permitiu gerar as classes PHP do modelo. Falta agora completar o arquivo **converteModelsEmTables.php** para que ele mande o Doctrine gerar as tabelas da base de dados:

```
01  <?php
02
03  require_once("bootstrap.php");
04
05  Doctrine::createTablesFromModels('../app/models/');
06
07  echo "As tabelas do banco foram geradas.";
```

Conforme vimos no esquema mostrado na sessão 7.3.1, o Doctrine só é capaz de gerar as tabelas da base de dados diretamente a partir das classes PHP do modelo. Por isso, vamos usar aqui, na linha 5, o método estático **createTableFromModels** da classe **Doctrine** passando para ele o diretório onde se encontram os arquivos que fazem parte do modelo da aplicação como parâmetro.

Para efeito ilustrativo, a execução do código acima significará para a base de dados o mesmo efeito que a execução do seguinte código SQL teria:

```
CREATE TABLE clientes (
   id BIGINT AUTO_INCREMENT,
   nome VARCHAR(255),
   email VARCHAR(255),
   PRIMARY KEY(id)
) ENGINE = INNODB;
```

## 4.4 Operações com as entidades

### *4.4.1 Visão Geral*

Esta sessão têm a finalidade de mostrar com funciona a criação, a leitura, a atualização e a exclusão de entidades utilizando o Doctrine. Para realizar cada uma dessas operações criaremos um novo arquivo PHP dentro da pasta pública da aplicação, a pasta ***www***.

Vamos chamar este arquivo onde iremos realizar alguns testes com o Doctrine de **testandoDoctrine.php**.

### *4.4.2 Salvando um novo cliente*

Para começar, vamos criar um novo cliente e pedir para persistí-lo na base de dados:

```
01  <?php
02
03  require_once("bootstrap.php");
04
05  require_once("../app/models/generated/BaseCliente.php");
06  require_once("../app/models/Cliente.php");
07
08  $cliente = new Cliente();
09  $cliente->nome = "Evandro Mazuco";
10  $cliente->email = "mazuco@septograma.com.br";
11  $cliente->save();
```

Na linha 3 carregamos o arquivo de *bootstrap* para montar o ambiente necessário para se trabalhar com o *framework* Doctrine. Na linha 5 e 6 carregamos as classes relacionadas a entidade cliente. Depois de instanciar um novo cliente (linha 8) e definir seu nome e email (linhas 9 e 10), pedimos para a entidade ser persistida através de uma chamada ao seu método **save**, herdado de **Doctrine_Record**.

### *4.4.3 Recuperando um cliente persistido*

Uma forma de fazer pesquisas na base de dados através do Doctrine é utilizando o **Doctrine_Query** da seguinte forma:

```
01  <?php
02
03  require_once("bootstrap.php");
```

```
04
05  require_once("../app/models/generated/BaseCliente.php");
06  require_once("../app/models/Cliente.php");
07
08  $consulta = Doctrine_Query::create()
09                          ->from('Cliente c')
10                          ->where('c.id = ?', 1);
11
12  $clientes = $consulta->execute();
13  $cliente = $clientes->get(0);
14  echo $cliente->nome;
```

Nas linhas 8 criamos um objeto **Doctrine_Query**, na linha 9 informação que esta consulta irá envolver a entidade cliente e na linha 8 especificamos que o cliente deverá ter seu valor de id igual a 1.

Na linha 12 executamos a consulta e o resultado apontado agora pela variável **$clientes**. Mesmo que a pesquisa tenha retornado apenas um cliente como resultado, o método **execute** de **Doctrine_Query** sempre retornará uma lista, daí a importância de, na linha 13, pegarmos o primeiro e único cliente que existe na lista. A linha 14 apenas exibe o nome do cliente.

### *4.4.4 Atualizando um cliente*

O código que iremos mostrar a atualização de um cliente é o que segue:

```
01  <?php
02
03  require_once("bootstrap.php");
04
05  require_once("../app/models/generated/BaseCliente.php");
06  require_once("../app/models/Cliente.php");
07
08  $consulta = Doctrine_Query::create()
09                          ->from('Cliente c')
10                          ->where('c.id = ?', 1);
11
12  $clientes = $consulta->execute();
13  $cliente = $clientes->get(0);
14
15  $cliente->nome = "Evandro Mazuco de Souza";
16  $cliente->save();
```

O processo envolve a necessidade de recuperar o cliente da base de dados (linhas 8 a 13), que já vimos na sessão anterior. Em seguida, na linha 15, alteramos o nome do cliente e mandamos persistir a alteração na linha 16.

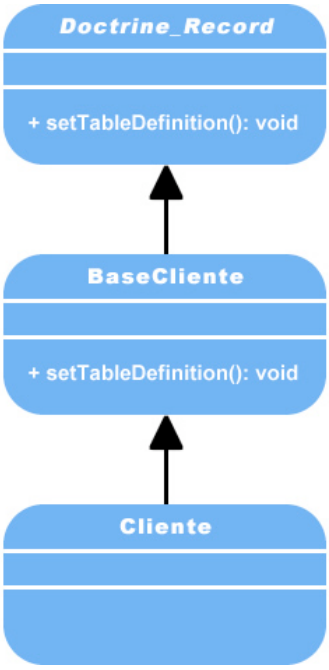
### *4.4.5 Excluindo um cliente*

Excluir um cliente também é bastante simples. Basta realizar a pesquisa na base de dados como já vimos e, em seguida, chamar o método **delete** da entidade, conforme mostrado na linha 15:

```
01  <?php
02
03  require_once("bootstrap.php");
04
05  require_once("../app/models/generated/BaseCliente.php");
06  require_once("../app/models/Cliente.php");
07
08  $consulta = Doctrine_Query::create()
09                          ->from('Cliente c')
10                          ->where('c.id = ?', 1);
11
12    $clientes = $consulta->execute();
13    $cliente = $clientes->get(0);
14
15    $cliente->delete();
```

## 4.5 Toda Entidade Herda de Doctrine_Record

Fazendo uma análise mais apurada, rapidamente iremos perceber que a classe **Cliente**, localizada na pasta ***models***, herda da classe **BaseCliente**, localizada na pasta ***generated***, e **BaseCliente**, por sua vez, herda da classe abstrata **Doctrine_Record** que faz parte da biblioteca do Doctrine.




A **Doctrine_Record** também é filha de mais um conjunto de classe e implementa uma série de interfaces. Com esta estrutura, a classe **Cliente** acaba ganhando "super-poderes". Poderes estes que impediram nossa implementação no Septo Framework implementasse de maneira mais coerente o padrão de projeto *Tree Layers Architecture*.

Isso aconteceu porque os objetos relacionados às entidades trafegam livremente dentro da arquitetura indo de um extremo ao outro e, como toda entidade herda de **Doctrine_Record**, ao fazer isso, carregam consigo funções que podem ser não compatíveis com o contexto onde possam se encontrar.

Por exemplo, não podemos fazer nada a respeito quando, na própria camada de apresentação, o programador resolve fazer uma chamada a um dos métodos da entidade herdados de **Doctrine_Record** lançando uma ação que terá repercussões no outro extremo da aplicação, a base de dados.

Nossa intenção inicial era a de que estas camadas – apresentação, negócio e persistência – comunicassem-se somente através de uma interface clara e específica, mas infelizmente, para utilizar o framework Doctrine no seu estágio de evolução atual[7], tivemos de abrir mão do rigor para fazer a arquitetura

[7] Antes do lançamento da versão 2.0 (ver trabalhos futuros)

funcionar e postergamos a expectativa de tornar as três camadas do Septo Framework o mais independentes e intercambiáveis possíveis para o lançamento das próximas versões do Doctrine.

# 5. SEPTO FRAMEWORK

## 5.1 Motivação

Desenvolver aplicações *web* de forma mais pragmática com uma ferramenta simples o suficiente para ser utilizada por principiantes e, ao mesmo tempo, poderosa a ponto de atender a necessidade dos desenvolvedores experientes foi o que nos levou a iniciar a elaboração do Septo Framework.

É comum para os estudantes de Sistemas de Informação observar o descompasso entre teoria e prática relacionados ao desenvolvimento de software. De um lado existe a academia, onde aprende-se programação orientada a objetos, engenharia de software, análise e projeto de sistemas, gerência de projetos e etc; do outro lado, também conhecido como o mundo real, um número incontável de projetos estourando prazos e orçamentos, gerando clientes insatisfeitos e desenvolvedores frustrados.

Mesmo parecendo difícil este casamento, aliar teoria e prática não é uma tarefa impossível. Estudo e aquisição de experiência, somado a muita paciência e trabalho persistente, foi uma fórmula que nos ajudou a concluir este projeto que já existe há quase 2 anos, o Septo Framework.

Poderíamos dizer que esta ferramenta é resultado de uma visão *bottom-up* no sentido de oferecer maturidade aos times de desenvolvimento, pois ela foca primeiro na forma de se programar e não na forma de gerenciar projetos. Justamente por isso, procurarmos criar uma ferramenta que possibilite que boas práticas de programação sejam incorporadas ao software final mantendo o foco na arquitetura e nos padrões de projetos utilizados.

Este é um projeto que evoluiu e transformou-se muito durante todo esse tempo mas que, no final, manteve sua proposta inicial: facilitar e dar suporte ao ciclo de desenvolvimento de software orientado a objetos para web através de uma ferramenta que seja capaz de aproximar teoria e prática no sentido de promover a maturidade e eficiência de times de desenvolvimento que utilizam PHP como linguagem.

## 5.2 Arquitetura Geral dos Sistemas Desenvolvidos com o Septo Framework

O Septo Framework possui três partes ou camadas. São elas: Apresentação, Negócio e Persistência. O Zend Framework não obriga a divisão do sistema dessa forma, muito menos o Doctrine. A implementação do padrão de projeto Tree Layers Architecture foi uma decisão de projeto específica do Septo Framework.

Estas três camadas estão distribuídas entre o *front-end* e o *back-end* das aplicações, estando a camada de Apresentação no *Front-end* e as camadas de Negócio e Persistência no *back-end*, conforme mostra a figura abaixo:


Esta divisão proporciona inúmeras vantagens como, por exemplo, a possibilidade de ter pessoas ou times separados realizando trabalhos em paralelo, cada um responsável por alguma parte específica da aplicação.

Outra vantagem é o fraco acoplamento entre as partes, uma vez que cada uma implementa sua respectiva interface. Assim, a realização de alterações em uma parte específica da aplicação não implica necessariamente o redesenho de outras.

Tem-se ainda, como vantagem de utilizar uma arquitetura estruturada desta forma, clareza naquilo que são as regras do negócio e onde elas devem estar materializadas em forma de código dentro das aplicações. Isso facilita tanto a manutenção da interface, que é parte da Camada de Apresentação, quanto a da base de dados na Camada de Persistência.

## 5.3 Foco na Entidades e no o Canal de Comunicação

O Septo Framework foi construído para facilitar e organizar a criação e manutenção das entidades de um sistema. Uma entidade é um objetivo

que pode representar todo tipo de abstração, como: um cliente, uma venda, um animal e etc.

Criar um *framework* focado em entidades significou materializar uma arquitetura onde é possível conseguir todo tipo de informação a respeito de uma entidade ou um conjunto de entidades através de um canal.

Este canal envolve nossa três camadas: apresentação, negócio e persistência. Ele é, portanto, a materialização do padrão de projeto *3 Tiers Architecture* em nossa arquitetura.

Sendo assim, quando é necessário, por exemplo, descobrir a data de nascimento de um cliente, uma solicitação é realizada através da interface do sistema que faz parte da camada de apresentação. Ainda na camada de apresentação, uma determinada *action* será executada dentro de algum *controller* específico. Esta *action* deve fazer uma invocação para um objeto da camada de negócio. A partir deste ponto, o objeto da camada de negócio assume e a action da camada de apresentação fica aguardando uma resposta. O objeto da camada de negócio, para descobrir a data de nascimento de um cliente, necessita carregar este cliente que está armazenado em uma base de dados e, para isso, faz uma invocação à camada de persistência. A camada de persistência, por sua vez, tem a responsabilidade de fazer a pesquisa na base e retornar a entidade correspondente ao cliente que a camada de negócios solicitou.

Quando a camada de persistência termina de fazer seu trabalho, o canal que envolve as três camadas do sistema segue seu fluxo em sentido contrário. Agora, a camada de persistência retorna um resultado para a camada de negócio e esta, por sua vez, retorna o cliente para a camada de apresentação que deve receber a resposta e exibir o resultado contendo a data de nascimento do cliente para o usuário do sistema.

## 5.4 Estrutura de diretórios do Septo Framework

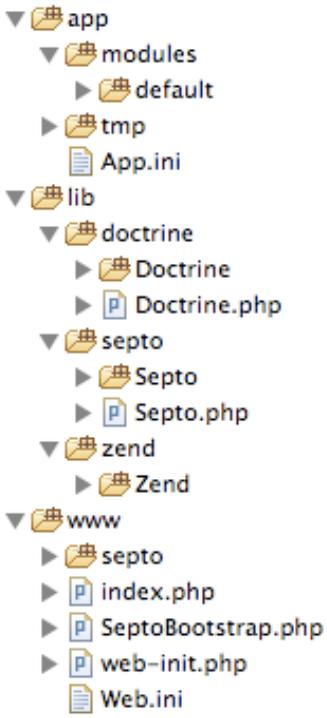
Nesta sessão, mais do que apenas apresentar a estrutura de diretórios do Septo Framework, mostraremos também onde estão os arquivos de configuração e qual a função de cada um deles.

Uma vez entendido a estrutura de diretórios modular apresentada no capítulo 6, onde falamos do Zend Framework, será tranquilo compreender a

organização do Septo Framework pois este utiliza exatamente a mesma estrutura modular.

A diferença ficou mais por conta do processo de inicialização, onde passou a existir uma classe de *bootstrap*, a SeptoBootstrap, e também uma série de recursos estendidos do Zend Framework que possibilitam o correto carregamento do Septo Framework.

É possível utilizar o Septo Framework com ou sem os Septo Modules instalados. Como vamos tratar dos Septo Modules somente no capítulo seguinte, vamos nos concentrar aqui em apresentar a estrutura mais enxuta possível para o Septo Framework, por enquanto, sem nenhum Septo Module:




Como se pode observar a estrutura é muito familiar à estrutura final apresentada no capítulo 6, com algumas pequenas adaptações. As pasta principais continuam sendo a pasta pública, a *www*, a pasta onde estão nossas bibliotecas, que é a *lib*, e a pasta onde fica todo o código principal da aplicação, a *app*.

Dentro da pasta *www* a novidade é a presença da classe **SeptoBootstrap**, do arquivo de inicialização **web-init.php** e do arquivo de configuração **Web.ini**.

Na pasta *lib* ouve um sutil mudança. Como o Doctrine, o Zend Framework e o próprio Septo Framework são obrigatórios para que se possa carregar o ambiente do Septo Framework, cada um destes *frameworks* ganhou cada uma pasta específica dentro de ***lib***:

- ***lib/doctrine*** para o Doctrine;
- ***lib/septo*** para o Septo Framework;
- ***lib/zend*** para o Zend Framework.

Isso não obriga o desenvolvedor a criar uma outra pasta especial dentro de ***lib*** para cada *framework* que resolver utilizar. A idéia é servir apenas para definir um local específico para colocar aqueles bibliotecas cruciais para o funcionamento do Septo Framework, ou seja: Zend Framework, Doctrine e o próprio Septo Framework.

Na pasta *app* continuou existindo a pasta *modules*, local onde ficam cada um dos módulos da nossa aplicação. Além disso, também inserimos ali uma outra pasta, a ***tmp***, que será utilizada pelo Septo Framework para, dentre outras coisas, armazenar os arquivos de *cache*.

Por fim, dentro de cada um dos *modules* – como no *module* Blog, por exemplo – passou a existir um arquivo de inicialização chamado **module-init.php** e um arquivo de configuração, o **Module.ini**.

Apresentaremos cada um desses arquivos e falaremos de sua utilidade a seguir, ao tratar do processo de inicialização do ambiente Septo Framework.

## 5.5 Inicialização do Septo Framework

### *5.5.1 O arquivo index.php*

A inicialização do Septo Framework começa no arquivo **index.php** contindo na pasta pública cujo código é o que segue:

```
01  error_reporting(E_ALL|E_STRICT);
02  ini_set("display_errors", "on");
03
04  try {
05      require_once "SeptoBootstrap.php";
06      $bootstrap = new SeptoBootstrap();
07      $bootstrap->dispatch();
08  } catch (Exception $e) {
09      echo $e->getMessage();
10  }
```

Tudo que o arquivo faz é pedir para exibir todos os possíveis erros, criar uma instância de SeptoBootstrap e mandá-la trabalhar. É a classe SeptoBootstrap que se encaregará de carregar o ambiente do Septo Framework e, caso algo inesperado aconteça, lançar uma exceção.

### *5.5.2 O arquivo Web.ini*

O SeptoBootstrap, embora esteja na pasta ***www*** – e não na pasta ***lib/septo*** –, é uma classe do Septo Framework e não deve ser alterada. No entanto, seu comportamento pode variar de acordo com o conteúdo do arquivo de configuração **Web.ini**. Abaixo, mostramos a configuração padrão contida em **Web.ini**:

```
01  [DEVELOPMENT]
02  DEBUG               = true
03
04  [APPLICATION]
05  TIMEZONE            = America/Sao_Paulo
06
07  [PATHS]
08  ;pasta publica (www) como referencia
09  APPLICATION         = ../app
10  LIBRARY             = ../lib
11
12  [INCLUDE_PATH]
13  SEPTO               = ../lib/septo
14  ZEND                = ../lib/zend
15  DOCTRINE            = ../lib/doctrine
```

Este arquivo de configuração possui uma sintaxe totalmente diferente do arquivo de configuração **YAML** que foi visto no capítulo 6. Internamente, o Zend Framework utilizará a função ***parse_ini_file***, uma função

nativa do PHP, para converter este arquivo de configuração em um *array* e, posterirmente, em um objeto **Zend_Config_Ini**.

Contudo, o importante para o usuário do Septo Framework é saber o que cada linha do arquivo acima faz deixando para o Septo Framework o compromisso de carregar o ambiente da aplicação.

Este arquivo é composto de cláusulas e cada cláusula começa numa nova linha com uma palavra entre colchetes. No arquivo acima temos três cláusulas, são elas: **DEVELOPMENT**, **APPLICATION**, **PATHS** e **INCLUDE_PATH**. Dentro de cada cláusula, encontram-se os parâmetros.

Na cláusula **DEVELOPMENT** há apenas um parâmetro, o **DEBUG**. Quando seu valor é *true* os erros que surgirem após o carregamento da classe **SeptoBootstrap** continuarão a serem exibidos no *browser*. Além disso, e o Septo Framework também passa a zerar o conteúdo em cache para cada nova solicitação do cliente com o objetivo de garantir que as alterações realizadas pelo desenvolvedor repercutam na aplicação.

A cláusula **APPLICATION** também contém apenas um parâmetro, o TIMEZONE. É boa prática lembrar de definir o **TIMEZONE** do Zend Framework, principalmente quando não se está nos EUA. Os recursos de localização do Zend Framework, por exemplo, irão tomar como base esta cláusula para definir aspectos relacionados a fuso-horário, formato de datas e moedas, por exemplo.

Embora sugerimos que as pastas ***app*** e ***lib*** estejam no mesmo nível da pasta ***www***, dependendo da decisão do desenvolvedor, isso poderá mudar. A sessão **PATHS** tem justamente o objetivo de permitir isso e, caso o desenvolvedor resolva colocar a pasta ***app*** ou ***lib*** em local diferente, basta alterar os valores dos parâmetros **APPLICATION** e **LIBRARY**.

Por fim, na cláusula **INCLUDE_PATH** é possível definir quais diretórios que devem ser incluídos no *path* da aplicação. A configuração *default* coloca as pastas onde estão o Doctrine, o Zend Framework e o próprio Septo Framework no *path*.

### *5.5.3 O arquivo web-init.php*

Depois de carregar todo o ambiente do Septo Framework, a última coisa que a classe **SeptoBootstrap** faz é ler o conteúdo do arquivo **web-init.php**.

Podemos inserir neste arquivo tudo aquilo que deve fazer parte do contexto da aplicação que não é possível fazer através do arquivo de configuração **Web.ini**.

Realizar a leitura de **web-init.php** é a última coisa que o Septobootstrap faz por isso, quando se esta no contexto de **web-init.php**, já é possível chamar as classes do Doctrine, do Zend Framework, do Septo Framework e tudo mais que já tenha sido colocado ou que possa ser entrado através do *path*.

Trata-se de um espaço onde o desenvolvedor pode, por exemplo, definir as regras de acesso da aplicação estendendo **Zend_Acl** e **Zend_Auth** sem se preocupar com os aspectos mais fundamentais relacionados ao carregamento do ambiente da aplicação, trabalho que já foi realizado pelo **SeptoBootstrap**.

### *5.5.4 O arquivo App.ini*

O **SeptoBootstrap** em conjunto com o arquivo **Web.ini** oferece recursos interessantes, mas seu trabalho principal é deixar tudo organizado e pronto para ser utilizado. Com o trabalho do **SeptoBootstrap** realizado, todo o ambiente da aplicação já estará pronto para funcionar utilizando Zend Framework e Doctrine de maneira integrada sem a necessidade de ficar "batendo-cabeça".

O Septo Framework foi criado para trabalhar exclusivamente com o recurso de múltiplos *modules* do Zend Framework ativado. Por isso, o caminho natural da aplicação depois de executar o *bootstraping* na pasta ***www*** será ir até a pasta onde estão os *modules* com o objetivo de localizar o conjunto *module-controller-action* que deve atender a uma determinada requisição.

Mas antes de entrar em um módulo específico, há ainda um arquivo dentro da pasta ***app***, o **App.ini**, que também é lido por **SeptoBootstrap**. Neste arquivo também estão definições que irão ter reflexo em toda a aplicação e sua configuração básica é a seguinte:

```
01   ; Database default para toda a aplicacao
02   [DATABASE]
03   HOST = 127.0.0.1
04   NAME = db_septo
05   USER = root
06   PASS = root
07   PORT =
```

```
08
09    [PARAMS]
10    estudante = Lucas Pacheco Teixeira
```

Este arquivo contém uma sessão para colocar informações referentes à base de dados padrão para todos os módulos. Ele também tem uma sessão voltada para a definição de parâmetros que serão visíveis em todos os *modules* da aplicação.

### *5.5.5 O arquivo Module.ini*

Dentro de cada um dos modules existe outro arquivo de configuração específico chamado **Module.ini** onde também existe uma sessão para colocar informações necessárias para acessar a base de dado e outra para definição de parâmetros.

Dessa forma, se ambas as configurações forem preenchidas, prevalecerá a configuração do arquivo **Module.ini**, uma vez que esta é mais específica do que a configuração da aplicação, o arquivo **App.ini**.

## 5.6 Entendendo o Funcionamento de uma Entidade

Para que uma entidade exista no Septo Framework é necessário um conjunto de arquivos específicos que irão preencher os espaços previstos na camada de apresentação, negócio e persistência.

Como exemplo ilustrativo, vamos tratar de analisar os arquivos necessários para se trabalhar com uma entidade chamada Cliente. A entidade Cliente será a mesma do exemplo usado no capítulo com o Doctrine, portanto, um cliente irá possuir um identificador, um nome e um email.

Para este exemplo, vamos considerar que estamos trabalhando dentro de um módulo chamado empresa. Como este é um módulo que utiliza os recursos do Septo Framework, ele possui os seguintes diretórios: ***configs***, ***controllers***, ***models***, ***persistence***, ***views*** e ***tmp***.

Já vimos com qual camada cada tipo de diretório está relacionado em nossa arquitetura, agora vamos saber diretório-por-diretório quais arquivos devem estar presentes para atender às necessidades de uma entidade e conhecer seu conteúdo.

### *5.6.1 Diretório configs*

Para qualquer entidade presente no Septo Framework é necessário que exista uma descrição desta entidade no arquivo **entities.yml** presente na pasta *configs*.

Utilizaremos este arquivo para descrever no entidade Cliente com seus três atributos propostos, a configuração do arquivo **entities.yml** deve ficar da seguinte maneira:

```
01  Cliente:
02    tableName: clientes
03    columns:
04      id:
05        type: integer(8)
06        primary: true
07        autoincrement: true
08      nome: string(255)
09      email: string(255)
```

### *5.6.2 Diretório controllers*

No diretório *controllers*, para atender as solicitações relacionadas a nossa entidade cliente, será necessário uma classe de controller em uma arquivo chamado **ClienteController.php**.

Para garantir a manipulação básica da entidade cliente (criar, listar, editar e excluir) tudo que este **controller** deverá fazer é herdar as características de **Septo_Entity_Controller**.

Dessa forma, com muito pouco código já temos condições de trabalhar com as entidades. Observe como fica a classe de *controller* para este caso:

```
01  <?php
02
03  class empresa_ClienteController extends Septo_Entity_Controller {
04
05  }
```

A classe **Septo_Entity_Controller** trata de implementar as actions mais básicas para a entidade Cliente, são elas: ***list***, ***create***, ***edit***, ***save*** e ***delete*** que podem ser acessadas conforme tabela abaixo:

| **Action** | **Url** (sem considerar o endereço principal do site) |
|---|---|
| *list* | /empresa/cliente/list |
| *create* | /empresa/cliente/create |
| *edit* | /empresa/cliente/edit |
| *save* | /empresa/cliente/save |
| *delete* | /empresa/cliente/delete |

Mesmo com o *controller* de **Cliente** criado, não será possível ainda testar as ações acima pois nos falta, dentre outras coisas, as classes PHP do modelo.

### *5.6.3 Diretório models*

As classes PHP do modelo deverão estar neste diretório. O diretório *models* possui 4 arquivos fundamentais para trabalharmos com a entidade Cliente. Três deles específicos para Cliente – os quais iremos mostrar em detalhes aqui – e um arquivo correspondente à fachada do modelo que é útil para tudo que estiver dentro da pasta ***models***.

Toda entidade da aplicação, portanto, deve possuir três arquivos específicos básicos. Para a entidade Cliente estes arquivos e seus diretórios serão os seguintes:

| **Arquivo** | **Diretório** | **Descrição** |
|---|---|---|
| BaseCliente | /empresa/models/entities/generated | Normalmente gerado através do Doctrine. É o arquivo que possui o código PHP que mapeia a entidade. |
| Cliente | /empresa/models/entities | Arquivo que estende **BaseCliente**. |
| ClienteService | /empresa/models/services | Arquivo que estende **Septo_Service** e centraliza as ações relacionadas a entidade. É quem realiza o negócio. |

O conteúdo do arquivo **BaseCliente.php** apresentado abaixo pode parecer um tanto confuso a primeira vista comparado ao que já vimos até aqui. Por isso, é saber que esta classe pode ser gerada automaticamente pelo Doctrine com base no arquivo de configuração e tudo que ela faz é colocar em forma de código PHP as mesmas definições presentes no arquivo de configuração YAML:

```
01  <?php
02
03  /**
04   * This class has been auto-generated by the Doctrine ORM Framework
05   */
06  abstract class BaseCliente extends Septo_Entity
07  {
08    public function setTableDefinition()
09    {
10      $this->setTableName('clientes');
11      $this->hasColumn('id', 'integer', 8, array('type' => 'integer',
         'primary' => true, 'autoincrement' => true, 'length' => '8'));
12      $this->hasColumn('nome', 'string', 255, array('type' => 'string',
         'length' => '255'));
13      $this->hasColumn('email', 'string', 255, array('type' => 'string',
         'length' => '255'));
14    }
15  }
```

Para a classe **Cliente**, bastará estender **BaseCliente**:

```
01  <?php
02
03  /**
04   * This class has been auto-generated by the Doctrine ORM Framework
05   */
06  class Cliente extends BaseCliente
07  {
08
09  }
```

E a classe **ClienteService**, para permitir que as ações básicas relacionadas a entidade possam ser executadas, deve estender **Septo_Service** e implementar os métodos ***__populate*** e ***__validate*** conforme mostrado abaixo:

```
01  <?php
02
03  class ClienteService extends Septo_Service {
04
05    public function __populate($id, Zend_Controller_Request_Abstract $request)
```

```
06    {
07      $cliente = $this->getEntity($id);
08      $cliente->nome = $request->getParam("nome");
09      $cliente->email = $request->getParam("email");
10
11      return $cliente;
12    }
13
14    public function __validate(Septo_Entity $cliente)
15    {
16      $messages = array();
17
18      if($cliente->nome == "")
19        $messages[] = "Nome do cliente deve ser informado.";
20
21
22      if($cliente->email == "")
23        $messages[] = "Email do cliente deve ser informado.";
24
25      return $messages;
26    }
27
28  }
```

Dentro do método **__populate**, cuja assinatura está na linha 5, recebemos um id como parâmetro para identificar a entidade um objeto do tipo **Zend_Controller_Request_Abstract** que nos oferece, dentre outras informações, os parâmetros que chegaram através de uma solicitação.

A primeira coisa que fazemos dentro desse método, na linha 7, é utilizar o método ***getEntity*** implementado em **Septo_Entity** passando o identificador recebido como parâmetro. Este identificador poderá vir vazio e isso irá significar que trata-se uma entidade nova, ainda não persistida na base de dados. Caso venha preenchido, é porque o mesmo método está sendo usado para realizar uma alteração em uma entidade já existente.

Nas linhas 8 e 9 “setamos” os valores dos atributos de cliente conforme os valores recebidos na requisição. Na linha 11 é importante lembrar de retornamos a entidade.

O método ***__validate***, linhas 14 a 26, é onde se deve fazer todas as validações que o usuário achar necessário. Neste caso, apenas exigimos que o atributo nome e email tenham sido informados. Ao final, linha 25, retornamos um *array* de erros. Caso tudo esteja de acordo com as regras de validações inseridas neste método, o *array* de erros é retornado permanecendo vazio.

### *5.6.4 Diretório persistence*

No diretório ***persistence*** é onde ficam todos arquivos relacionados à camada de persistência. Esta camada é acessada a partir do modelo – a camada de negócio –, utilizando uma classe de fachada.

Independente dos arquivos específicos para cada uma das entidades da aplicação, o arquivo de fachada para a camada de persistência sempre deverá estar presente. Este arquivo fica diretamente dentro do diretório ***persistence*** e seu nome deve ser composto no nome do módulo seguido por "Persistence".

Como nosso módulo de exemplo chama-se empresa, o arquivo de fachada para acessar a camada de persistência irá se chamar: **EmpresaPersistence.php** e terá a seguinte configuração:

```
01  <?php
02
03  class EmpresaPersistence extends Septo_Persistence_Facade {
04
05  }
```

Além do arquivo da fachada, existem os arquivos *DAOs* –referência ao padrão de projeto Data Access Object – que compõem a camada de persistência. São esses arquivos que, de fato, fazem as consultas na base de dados. Eles ficam dentro de um diretório específico chamados ***daos*** que está dentro da pasta ***persistence***.

Cada entidade possui o seu respectivo arquivo DAO. Em nosso caso, onde temos apenas a entidade cliente, nosso único arquivo DAO será **ClienteDAO.php** localizado no seguinte diretório:

app/modules/empresa/persistence/daos/ClienteDAO.php

No Septo Framework existe um classe abstrata chamada **Septo_Persistence_DAO**. Esta classe está pronta para atender às operações básicas relacionadas a quaisquer entidades. Por isso, apenas estendendo

**Septo_Persistence_DAO** nossa classe **ClienteDAO** já estará pronta para ser utilizada em nosso exemplo:

```
01  <?php
02
03  class ClienteDAO extends Septo_Persistence_DAO {
04
05  }
```

### *5.6.5 Diretório views*

Os últimos arquivos que ainda precisam ser criados para que se possa trabalhar com a entidade Cliente estarão dentro da pasta ***views*** e, para cada uma das *actions* herdadas de **Septo_Entity_Controller** por **Empresa_ClienteController**, será necessário a criação do seu respectivo arquivo de *script*, que são arquivos com extensão **.phtml** que deverão estar localizados em:

app/modules/empresa/views/scripts/

As *actions* herdadas de **Septo_Entity_Controller** são:

1. ***createAction***
2. ***saveAction***
3. ***listAction***
4. ***editAction***
5. ***deleteAction***

Para atender estas *actions*, na *view* teremos de criar os seguinte arquivos de script, respectivamente:

1. **create.pthtml**
2. **save.phtml**
3. **list.phtml**
4. **edit.phtml**
5. **delete.phtml**

Para começar, como ainda não temos nenhum registro de cliente na base de dados, vamos criar o arquivo de *script* onde deve estar o formulário que será usado para inserir um novo cliente, o **create.phtml**:

```
<form action="/empresa/cliente/save" method="post">
 <fieldset>
  <legend>Novo Cliente:</legend>
  Nome: <input name="cliente_nome" type="text" value="">
  Email: <input name="cliente_email" type="text" value="">
  <input name="Salvar" type="submit" value="Salvar">
 </fieldset>
</form>
```

Este formulário, ao ser enviado, irá chamar a *action* ***saveAction***. Para esta *action* criaremos um segundo *script*, o **save.phtml** cuja função será exibir uma mensagem de confirmação e oferecer um *link* para lista as entidades Cliente:

```
<h1>Cliente salvo!</</h1>
<p><a href="/empresa/cliente/list">listar todos</a></p>
```

O *link* "Listar todos" do arquivo acima chamará a *action* ***listAction*** e, para esta *action* funcionar, teremos que criar mais um arquivo de *script*, o **list.phtml**:

```
<h1>Lista de Clientes...</h1>

<table border=1>
<? foreach($this->entidades as $entidade) { ?>
  <tr>
    <td><?= $entidade->id ?></td>
    <td><?= $entidade->nome ?></td>
    <td><?= $entidade->email ?></td>
    <td>
      <a href="/empresa/cliente/edit/cliente_id/<?= $entidade->id ?>">Editar</a>
    </td>
    <td>
      <a href="javascript:confirmarExclusao(<?= $entidade->id ?>);">Deletar</a>
    </td>
  </tr>
<? } ?>
</table>

<script type="text/javascript">
  function confirmarExclusao(id) {
```

```
21      if (confirm("Excluir cliente "+id+"?")) {
22        var local = "/empresa/cliente/delete/cliente_id/"+id;
23          location.href=local;
24      }
25    }
26  </script>
27
28  <p><a href="/empresa/cliente/create">Criar novo Cliente</a></p>
```

A função deste arquivo é listar as entidades Cliente existentes. Além disso, na linha 10 e 13, colocamos *links* para editar e excluir um cliente, respectivamente. O código javascript, entre as linhas 19 e 26, serve para exibir uma mensagem de confirmação antes de excluir um cliente. Na linha 28, oferecemos um *link* para o formulário de cadastro de novos clientes apenas para facilitar a navegação.

Outro *script* que ainda deve ser criado é o **edit.phtml**:

```
01  <form action="/empresa/cliente/save" method="post">
02   <fieldset>
03    <legend>Editando um Cliente:</legend>
04
05    Nome: <input name="cliente_nome" type="text"
06                 value="<? echo $this->entidade->nome; ?>">
07
08    Email: <input name="cliente_email" type="text"
09    value="<? echo $this->entidade->email; ?>">
10
11    <input name="cliente_id" type="hidden"
12    value="<? echo $this->entidade->id; ?>" />
13
14    <input name="Salvar" type="submit" value="Salvar" />
15   </fieldset>
16  </form>
```

O conteúdo de **edit.phtml** é bastante parecido com o de **create.phtml**. Na verdade, conhecendo um pouco mais dos recursos do Zend Framework é possível transformar estes dois arquivos em apenas um, mas vamos prosseguir com a criação do último arquivo de *script* que está faltando, o **delete.phtml**, cuja única função é mostrar uma mensagem de confirmação informando que o cliente foi apagado e oferecer um *link* para o usuário voltar mais facilmente à lista de clientes.

```
01  <h1>Cliente deletado!</h1>
02  <p><a href="/empresa/cliente/list">listar todos</a></p>
```

Completado estes procedimentos, temos tudo que iremos precisar para trabalhar com a entidade Cliente através do canal de comunicação. Apresentamos em detalhes a composição de cada um dos arquivos, mas isso não significa que para utilizar o Septo Framework todos eles deverão ser criados manualmente.

No próximo capítulo veremos um forma de gerar todos os arquivos necessários para compor o canal de comunicação sem ter que digitar nenhuma linha de código PHP, HTML ou Javascript, tudo isso graças às funcionalidades oferecidas pelos Septo Modules.

# 6. SEPTO MODULES

## 6.1 Conceito

Septo Modules são *modules* especiais feitos para serem utilizados por aplicações baseadas no Septo Framework cuja função é auxiliar no processo de desenvolvimento, administração e utilização dos sistemas criados.

A arquitetura apresentada no capítulo 8 é materializada nos Septo Modules, sendo eles mesmos exemplos de aplicações criadas utilizando a arquitetura proposta pelo Septo Framework.

Dessa forma, além de servirem para os mais diferentes objetivos que serão mostrado a serguir, a própria possibilidade de estudar o seu código-fonte os torna uma espécie de guia para construção de novos *modules* com as mais diferentes funcionalidades.

Hoje existem três Septo Modules. Dois deles concluídos, o Septo Panel e o Septo Cyclo, e outro em fase de conclusão, o Septo Access. Cada um deles possui funções bem específicas e foram desenvolvidos para facilitar o trabalho do desenvolvedor de sistemas web.


Neste capítulo mostraremos como instalar os Septo Modules e como configurá-los. Além disso, apresentaremos em detalhes o Septo Cyclo e o Septo Panel. Com relação ao Septo Access, ainda em fase de desenvolvimento, trataremos de mostrar qual o seu propósito e como se faz para utilizar sua versão de teste.

## 6.2 Septo Cyclo

### *6.2.1 Visão Geral*

É natural que o primeiro Septo Module de que vamos falar seja o Septo Cyclo, uma vez que ele é capaz de gerar todos os arquivos necessários para se trabalhar com as entidades conforme mostrado no capítulo anterior.

Para ter uma ideia do que isso significa em termos de redução de esforço, vamos tomar como exemplo a mesma entidade Cliente da qual falamos no capítulo anterior. Naquele momento, foram criados uma série de arquivos básicos para se trabalhar com a entidade Cliente através do canal de comunicação envolvendo as três camadas da arquitetura proposta.

Com exceção dos arquivos da pasta ***views***, todos os demais são arquivos criados naquele momento podem ser gerados automaticamente pelo Septo Cyclo. O Septo Cyclo também é capaz de gerar os seguintes arquivos na pasta de *scripts* da *view*: **_create.phtml**, **_list.phtml** e **_listhead.phtml**. Tais arquivos só serão úteis caso estejamos utilizando também outro Septo Module, o Septo Panel. Isso significa que utilizando estes dois Septo Modules é possível gerar todos os arquivos necessários para se trabalhar com uma entidades através do canal de comunicação sem escrever nenhuma linha de código PHP.

Vamos tratar da utilização conjunta do Septo Cyclo e do Septo Panel logo a seguir, por enquanto, o importante é compreender o papel do Septo Cyclo.

### *6.2.2 Instalação*

Para baixar o Septo Cyclo basta ir até a página do Septograma (www.septograma.com.br) e procurar pelo arquivo compactado contendo o Septo Cyclo. Ao descompactar o arquivo irá perceber que existiram duas pastas: ***SeptoCyclo*** e ***septo***.

A pasta ***SeptoCyclo*** deverá ser colocada no diretório onde ficam os modules:

```
app/modules/SeptoCyclo
```

A pasta ***septo*** deverá ser colocada diretamente na pasta pública, normalmente ***www***:

www/septo

Com o Septo Cyclo já instalado, fica faltando apenas informar quais *modules* devem utilizar o Septo Cyclo e quais deverão ignorá-lo.

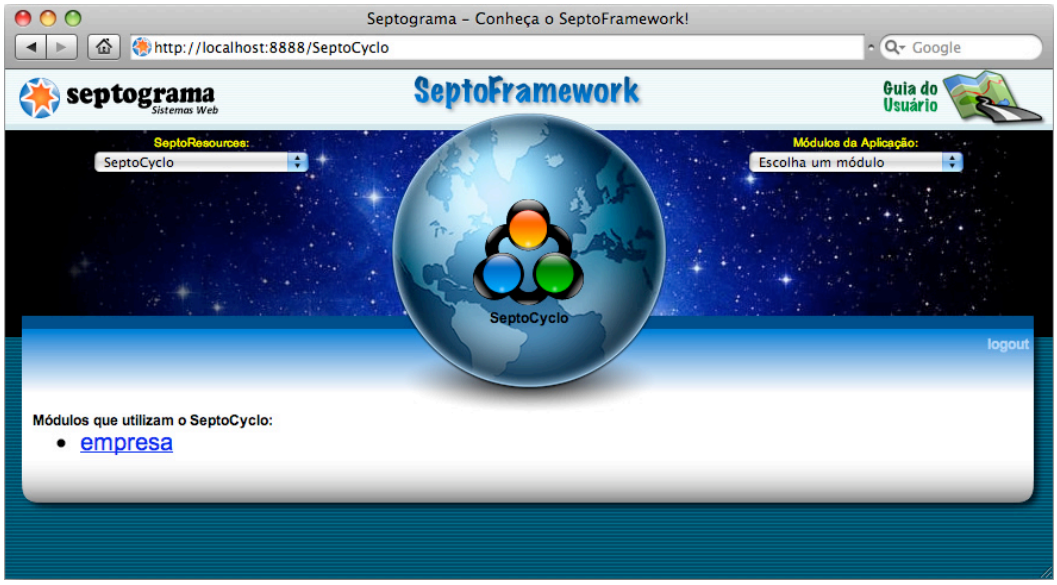
Isso é feito dentro do arquivo **Module.ini** de cada *module*. Vamos supor, por exemplo, que estejamos interessados em utilizar o Septo Cyclo com o *module* **empresa** criado nos capítulos anteriores. Temos então que ir até o arquivo **Module.ini** de **empresa**, procurar pela cláusula **SEPTO_RESOURCES** e inserir nela um parâmetro *SeptoCyclo* com o valor *true*. O arquivo abaixo mostra um arquivo **Module.ini** completo contendo a cláusula **SEPTO_RESOURCES** na linha 8 e o parâmetro SeptoCyclo com o valor true na linha 11:

```
01  [DATABASE]
02  HOST         = localhost
03  NAME         = base_empresa
04  USER         = root
05  PASS         = root
06  PORT         =
07
08  [SEPTO_RESOURCES]
09  SeptoPanel   = true
10  SeptoAccess  = true
11  SeptoCyclo   = true
```

Uma vez feito isso, podemos agora ir até o navegador web e começar a utilizar o Septo Cyclo através do seguinte endereço:

http://localhost/SeptoCyclo

Por temos definido que o *module* **empresa** passaria a utilizar o Septo Cyclo, teremos como resultado a seguinte tela de navegação:




Agora sim, o Septo Module já está pronto para ser utilizado pelo *module* **empresa**. Proseguiremos mostrando como é simples criar os arquivos necessários para se trabalhar com a entidade Cliente utilizando o Septo Cyclo.

### *6.2.3 Utilização*

Ao clicarmos no *link* empresa teremos acesso a seguinte tela:


Esta é tela principal do Septo Cyclo e nela podemos ver que esta ferramenta é capaz de realizar três tipos de operações que compõem um ciclo (daí a origem do seu nome):

1. Ler a base de dados e gerar o arquivo de configuração **entities.yml**;
2. Ler o arquivo de configuração **entities.yml** e gerar todo o código PHP para se trabalhar com uma entidade através do canal de comunicação;
3. Ler o cógido PHP e com base nele criar as tabelas do banco de dados.

Embora esta três operações componham uma espécie de ciclo, não faz sentido seguir uma caminho que vá da ação 1 até a ação 3. Em nosso caso, ao desenvolver sistemas web, preferimos partir de uma base de dados bem modelada e gerar o arquivo **entities.yml** a apartir dela, ou seja, começamos com a ação 1. Depois disso, através da ação 2, geramos o código PHP que irá compor

cada uma das camadas da aplicação com base no arquivo de configuração **entities.yml**. Com essa estratégia, a ação 3 acaba nunca sendo utilizada.

Fica a critério do desenvolvedor escolher o melhor caminho uma vez que o Septo Cyclo é capaz de oferecer diferentes estratégias de desenvolvimento. Contudo, neste momento, iremos seguir com a explicação do Septo Cyclo partindo de um exemplo onde já temos uma base de dados com uma tabela chamada *clientes*, a mesma base utilizada nos exemplos anteriores.

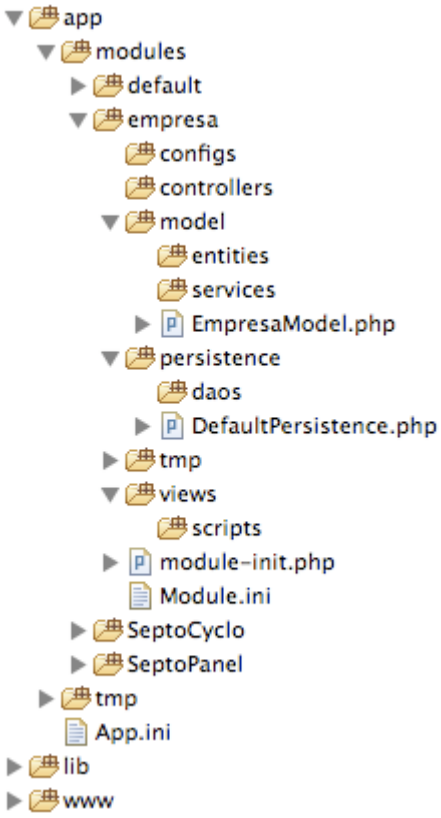
Logicamente, uma vez que já temos a base de dados, vamos continuar com o desenvolvimento do sistema utilizando primeiramente a ação 1, onde é possível gerar o arquivo de configuração **entities.yml** a partir da base de dados.

Para ilustrar o que irá acontecendo com a aplicação conforme vamos executando cada uma das ações propostas, vamos deixar claro em que ponto está nossa aplicação e o que irá acontecendo com ela conforme avançamos na utilização do Septo Cyclo.

Por enquanto, o que temos é uma base de dados onde existe uma tabela chamada *clientes* sem nenhum registro. Esta tabela foi criada a com a execução do seguinte código SQL:

```
CREATE TABLE clientes (
   id BIGINT AUTO_INCREMENT,
   nome VARCHAR(255),
   email VARCHAR(255),
   PRIMARY KEY(id)
) ENGINE = INNODB;
```

Também iremos considerar que a estrutura de diretórios da nossa aplicação encontra-se, nesse primeiro momento, da seguinte forma:




Observe que apenas existem os arquivos relacionados as fachadas da Camada de Negócio e da Camada de Persistência e falta a presença do arquivo **entities.yml** na pasta *configs*, além de todos os arquivos específicos de Cliente necessários para se trabalhar com esta entidade. O passo natural, uma vez que estamos partindo da base de dados, será navegar até o Septo Cyclo, escolher a opção "empresa" e clicar na seta que liga a imagem "DB TABLES" à imagem "YAML FILES".

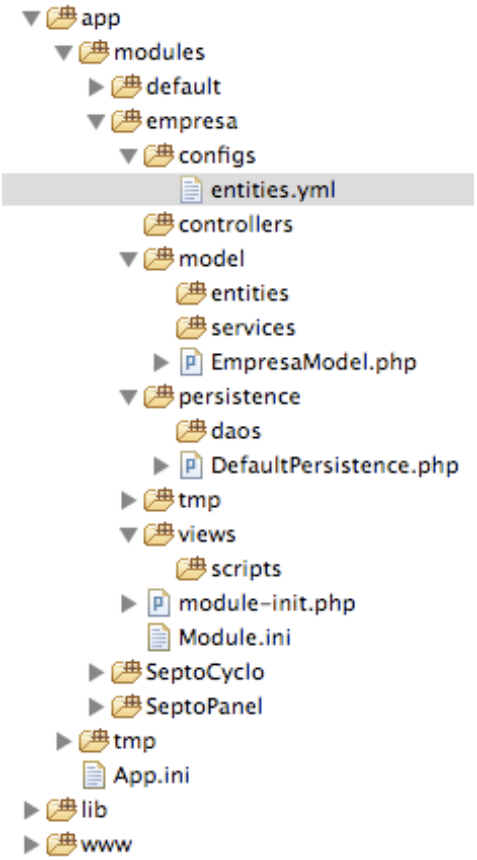
Com esta operação o arquivo **entities.yml** passará a existir, ficando da seguinte forma:

```
01  Clientes:
02    tableName: clientes
03    columns:
04      id:
05        type: integer(8)
06        primary: true
07        autoincrement: true
08      nome: string(255)
09      email: string(255)
```

Por padrão de comportamento, o arquivo **entities.yml** define uma entidade para cada tabela na base de dados e oferece para esta entidade o mesmo nome da tabela. Como nossa tabela trás um nome no plural, vamos fazer uma pequena alteração no arquivo **entities.yml** gerado para que o mesmo passe a considerar como nome de classe para a tabela *clientes* o nome Cliente, no singular:

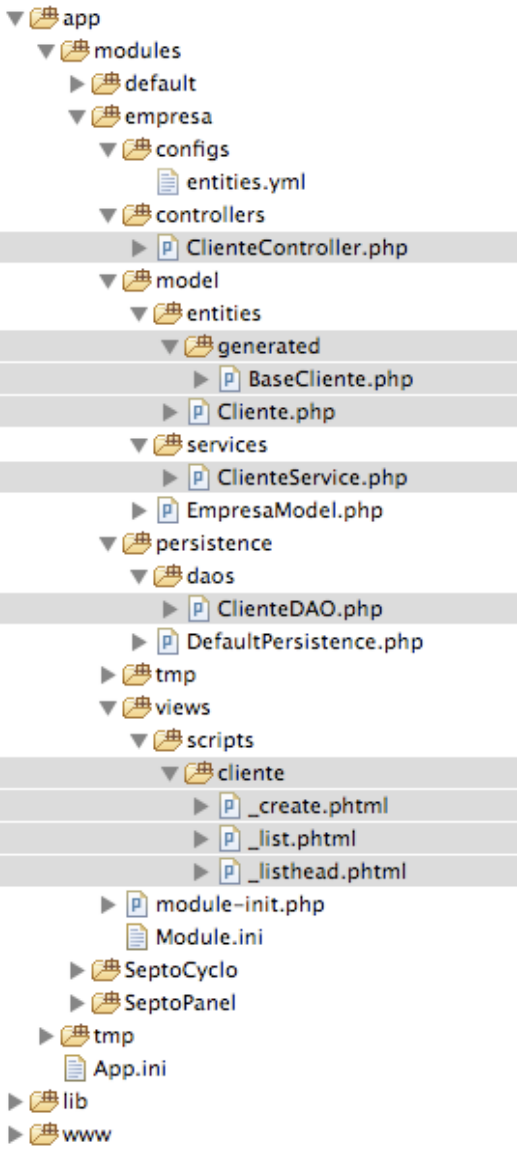
```
01  Cliente:
02    tableName: clientes
03    columns:
04      id:
05        type: integer(8)
06        primary: true
07        autoincrement: true
08      nome: string(255)
09      email: string(255)
```

Até aqui, tudo que mudou na aplicação foi a presença do arquivo **entities.yml** na pasta ***configs***:




Partimos agora para a ação 2, onde vamos gerar o código PHP necessário para trabalhar com a entidade Cliente. Para isso, basta utilizar a mesma tela principal do Septo Cyclo e clicar na seta que liga a imagem “YAML FILE” à imagem “PHP CODE”.

Após realizar tal operação, uma série de novos arquivos relacionados a entidade Cliente passarão a fazer parte da aplicação compondo cada uma das três camadas:




Neste ponto, bastaria apenas terminar o trabalho de construir todo o canal de comunicação criando os arquivos necessários para a *view*. Este trabalho é igual ao que foi realizado na sessão **5.7.5** do capítulo anterior. Lembrando que caso estejamos utilizando o Septo Cyclo em conjunto com o Septo Panel, tal tarefa sequer seria necessária. Por isso, a seguir partiremos da aplicação no ponto que ficou até aqui para apresentar o Septo Panel.

## 6.3 SeptoPanel

### *6.3.1 Visão Geral*

O Septo Panel dispensa o trabalho de criar um painel para realizar a administração das entidades. Este módulo carrega consigo um *layout* igual ao

do Septo Cyclo e mais um conjunto de *scripts* que tornam possível o trabalho de criar, apagar, editar e listar as entidades que foram criadas via *browser* sem na necessidade de criação de nenhum código extra além daquele gerado pelo Septo Cyclo.

A ideia é que a o Septo Panel possa trabalhar em conjunto com o Septo Cyclo com a finalidade de tornar mais rápido o desenvolvimento de sistemas *web* utilizando o Septo Framework, incluindo os casos em que temos a presença de entidades mais complexas, aquelas que têm algum tipo de relacionamento com outra(s) entidade(s). Abordaremos este assunto no próximo capítulo.

### *6.3.2 Instalação*

A instalação do Septo Panel dá-se da mesma forma que a instalação do Septo Cyclo. Basta ir até a página do Septograma (www.septograma.com.br) baixar o arquivo compactado referente ao Septo Panel. Ao descompactá-lo, duas pastas chamadas ***SeptoPanel*** e ***septo*** deverão ser colocadas em seus respectivos lugares dentro da aplicação.

A pasta ***SeptoPanel*** ficará no diretório onde se encontram os *modules*, mesmo nível onde colocamos o Septo Cyclo. A pasta ***septo*** deverá ficar na pasta pública da aplicação, normalmente ***www***.

A pasta ***septo*** do Septo Panel é a mesma pasta ***septo*** que acompanha o Septo Cyclo. Se a pasta ***septo*** de Septo Cyclo já tiver sido colocada no diretório público não há a necessidade de sobrescrevê-lo, uma vez que ambas possuem exatamente o mesmo conteúdo.

O última coisa que devemos lembrar é de ir até o arquivo **module.ini** daqueles módulos que devem utilizar o Septo Panel e procurar pela cláusula **SEPTO_RESOURCES**. Dentro desta cláusula deverá haver uma propriedade chamada *SeptoPanel* com o valor *true*, conforme mostrado na linha 9 do arquivo **module.ini** em nosso module **empresa**:

```
01  [DATABASE]
02  HOST         = localhost
03  NAME         = base_empresa
04  USER         = root
05  PASS         = root
```

```
06  PORT         =
07
08  [SEPTO_RESOURCES]
09  SeptoPanel   = true
10  SeptoAccess  = true
11  SeptoCyclo   = true
```

Seguido esses passos, o Septo Panel estará pronto para ser utilizado.

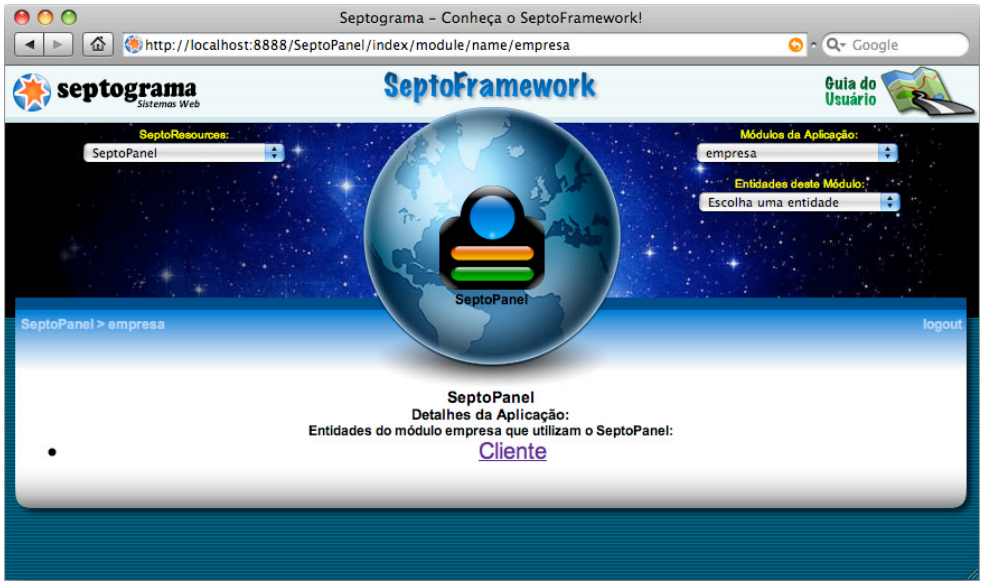
### *6.3.3 Utilização*

Uma vez instalado o Septo Panel, basta acessar uma url conforme segue:

http://localhost/SeptoPanel

Ao acessar esta *url*, a primeira tela que o usuário verá será aquela que mostra quais modules da nossa aplicação utilizam o Septo Panel. Como estamos trabalhando apenas com um *module* de exemplo chamado **empresa**, este é o único que aparece:


Selecionando o *module* **empresa** veremos que dentro dele há apenas uma entidade para ser acessada via o Septo Panel, a entidade Cliente:




Ao clicar em Cliente iremos diretamente para uma tela onde é exibido uma lista contendo os clientes existentes:


Como ainda não temos nenhum cliente na base de dados a lista está vazia. Para mudar isso, vamos clicar no botão “novo” no lado direito superior da tabela. Ao fazer isso, vamos para uma tela onde é possível cadastrar clientes:


Informamos nome e email do cliente e clicamos no botão salvar. Fazendo isso, a seguinte tela deverá exibida:


Ao listar, perceberemos que o cliente realmente foi salvo na base de dados e poderá ser agora excluído ou alterado:


Ao clicar no botão editar (representado por um lápis) o mesmo cliente reaparece no formulário pronto para ser editado. Ao clicar no botão excluir (representando por um "x" vermelho) aparece uma tela de confirmação perguntando se temos certeza de que a entidade deve ser excluída.

De maneira geral, a navegação pelo Septo Panel é bastante intuitiva. Mas ainda voltaremos nesse assunto no capítulo 10 para mostrar como o Septo Panel irá se comportar ao trabalhar com entidades que possuem algum tipo de relacionamento.

## 6.4 SeptoAccess

### *6.4.1 Visão Geral*

Dentre os Septo Modules existentes, o Septo Access é aquele que encontra-se em fase de conclusão. Seu função consiste em utilizar recursos da biblioteca do Zend Framework – **Zend_Acl** e **Zend_Auth** – para implementar um mecanismo automatizado de autenticação e controle de acesso.

Uma vez concluído, será possível configurar dinamicamente através do browser que tipo de recursos poderão ser acessados por determinados grupos de usuários.

O Septo Access completará a tríade de Septo Modules voltados para o desenvolvimento de aplicações *web* de forma rápida e com foco na qualidade do código gerado e na manutenabilidade dos sistemas.

Enquanto este Septo Module não fica pronto, a sugestão é conhecer bem as recursos **Zend_Acl** e **Zend_Auth** e implementá-los com base na documentação da própria **Zend**.

Lembramos que tais recursos precisão ser carregados no momento da inicialização, daí a importância da existência do arquivo **web-init.php** do Septo Framework, onde o desenvolvedor poderá incluir tudo aquilo que irá influenciar em toda a aplicação, incluindo o código necessário para ativar os recursos de controle de acesso proposto pelo Zend Framework em seus manuais.

# 7. EXEMPLO PRÁTICO

## 7.1 Visão Geral

Entendido os conceitos básicos que envolvem a utilização das ferramentas propostas neste trabalho, agora iremos partir para o desenvolvimento de um site completo com base no Septo Framework e seus Septo Modules.

O ambiente de desenvolvimento utilizado para criar esta aplicação envolverá:

- Mac OS X como sistema operacional;
- Safari como navegador web;
- Eclipse Galileu como IDE;
- MAMP como ferramenta para gerenciar nosso site localmente. O MAMP trás consigo o Apache, MySQL e PHP.

A aplicação que iremos fazer será de uma empresa fictícia, um Shopping Center composto apenas por lojas que compartinham um ideal de promover o consumo consciente.

O site deste shopping deve apresentar um conjunto de lojas na sua página principal, cada uma dessas lojas possui um conjunto de mensagens que podem colocar no site referentes as suas promoções, notícias, etc.

O objetivo com a apresentação deste exemplo prático é a de:

- Oferecer uma visão completa de como um sistema poderá ser desenvolvido utilizando o Septo Framework e seus Septo Modules;
- Compreender melhor o funcionamento do próprio Zend Framework e apresentar novos aspectos relacionados à sua camada de apresentação;
- Mostrar como é possível desenvolver web sites completos de forma rápida e prática com o Septo Framework;

- Criar *actions* que invocarão a camada de negócios e carregarão na *view* as entidades que ela precisa para construir as páginas do nosso site;
- Saber como definir relacionamentos através do arquivo de configuração YAML.
- Observar como o Septo Panel é capaz de criar um painel de administração de conteúdo dinâmico mesmo envolvendo entidades que possuem algum tipo de relacionamento;
- Servir como uma espécie de guia para a construção de novos sistemas.

Para alcançar todos esses objetivos não precisaremos de um domínio de negócio complexo, apenas duas entidades serão necessárias para dar esta visão geral, serão elas:

| Entidade | Descrição |
|---|---|
| Loja | Uma loja possui um nome, uma descrição, um telefone, um email e um conjunto de mensagens. |
| Mensagem | Uma mensagem pertence a uma loja e possui uma data e um conteúdo. |

Como estratégia de desenvolvimento, vamos continuar seguindo o caminho que primeiro desenvolve a base de dados, depois gera o arquivo de configuração YAML e, por último, constrói o código PHP necessário.

## 7.2 Construindo a aplicação

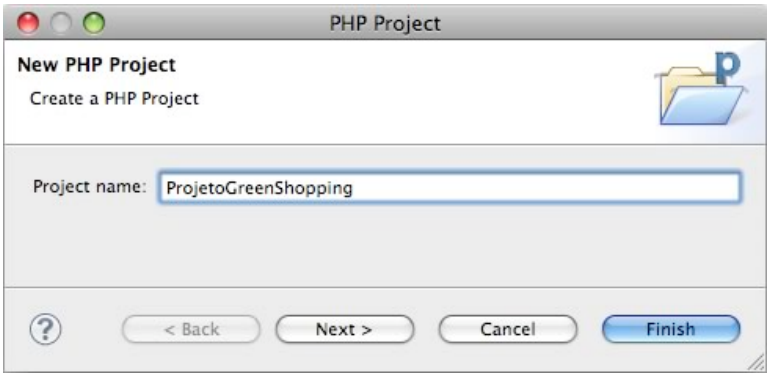
### *7.2.1 Criando a estrutura básica de diretórios*

Para começar, é preciso que tenhamos a estrutura básica de diretórios construída e todos os arquivos necessários para o funcionamento básico do Septo Framework em seus devidos lugares.

Faremos isso construindo um projeto no Eclipse Galileu utilizando um plugin para trabalhar com a linguagem PHP chamado PDT Project[8]. Para criar

[8] http://www.eclipse.org/pdt/

o nosso projeto iremos através do menu do Eclipse na opção  menu File > New > PHP Project conforme mostrar a figura abaixo:



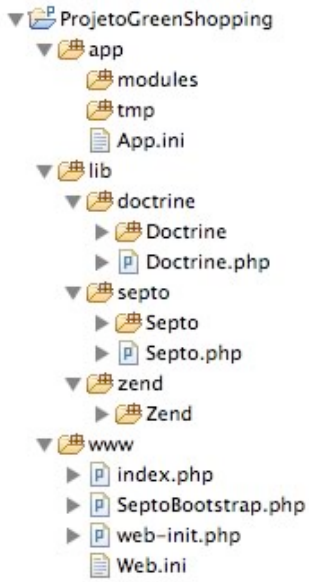

Dentro do projeto recém criado colocaremos os três diretórios básicos para rodar uma aplicação com o Septo Framework: ***app***, ***lib*** e ***www***.

Em ***www*** colocaremos os seguintes arquivos já abordados em capítulos anteriores: **index.php**, **SeptoBootstrap.php**, **.htaccess**, **Web.ini** e **web-init.php**.

Dentro de ***lib*** estarão três pasta específicas onde devem ficar as bibliotecas do Doctrine, do Septo Framework e do Zend Framework. Essas pastas irão se chamar ***doctrine***, ***septo*** e ***zend***, respectivamente. Feito isso, basta dirigir-se até o site de cada uma das ferramentas acima e descompactar os arquivos em seus respectivos diretórios.

Na pasta ***app*** deverá estar o arquivo de configuração **App.ini** e também outras duas pastas: a pasta ***modules***, onde estarão os módulos da nossa aplicação, e a pasta ***tmp***.

Até este ponto, o projeto deverá estar com a seguinte aparência:

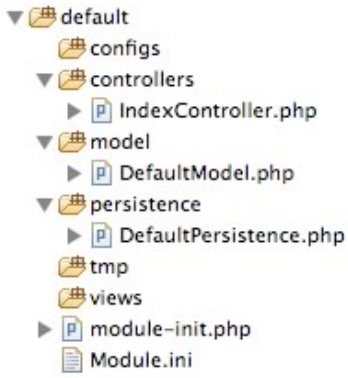



Observe que o arquivo **.htaccess** que está dentro na pasta www não é exibido pelo Eclipse (a não ser que troquemos sua configuração padrão).

### *7.2.2 Criando o module Default*

O *module* ***default*** será utilizado aqui com a intenção de exibir as principais páginas do site, aquelas que serão visualizadas pelos clientes do Shopping.

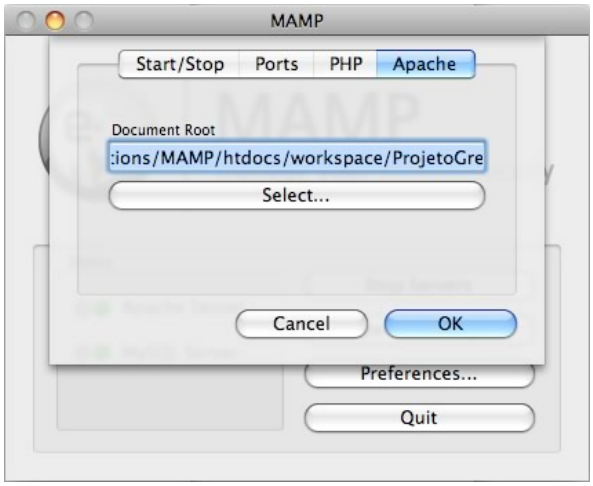
O *module* ***default*** será um módulo que irá utilizar os recursos do Septo Framework, por isso, precisa obrigatoriamente possuir um conjunto específico de pastas, arquivos de configuração e fachadas para acessar a camada de negócios e a de persistência, ficando ele da seguinte forma:



Também criamos na estrutura apresentada acima um *IndexController* para o *módulo* ***default*** e nele escreveremos a ***indexAction*** para testar se tudo está funcionando até aqui:

```
01  <?php
02
03  class IndexController extends Zend_Controller_Action {
04
05     public function indexAction() {
06      }
07
08  }
```

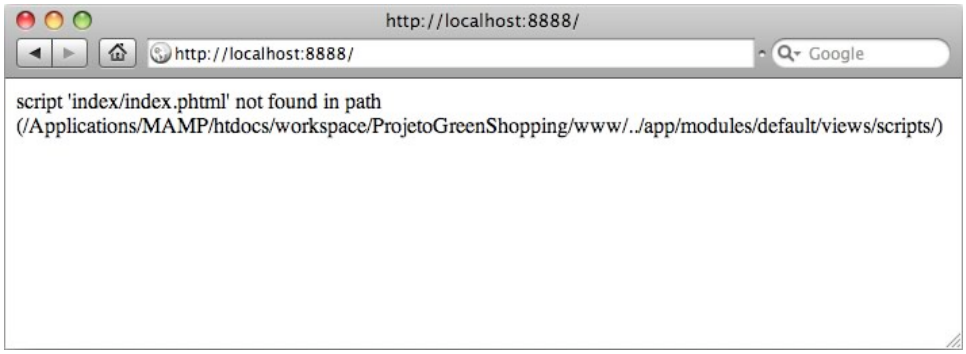
Antes de testar o site, precisaremos apontar o servidor apache para a pasta pública do projeto criado. Para fazer esta operação utilizando o MAMP basta abrir seu painel de administração, clicar em *preferences*, ir até a guia Apache, alterar a configuração do *Document Root* para apontar para a pasta *www* de nosso projeto e clicar em "ok", conforme mostrado abaixo:




Com o Apache inicializado e corretamente configurado, podemos abrir o navegador e digitar o seguinte endereço:

http://localhost/

Tal operação irá invocar a ***indexAction*** do **IndexController** do módule **default**. Neste momento, se tudo estiver certo até aqui, deverá aparecer a seguinte tela reportando que faltou criar os arquivos de *script* da *view*:




### *7.2.3 Criando os arquivos da View*

Além do arquivo de *script* necessário para fazer funcionar nossa *indexAction* temos que criar todo o *layout* do site e ativar este recurso do Zend Framework.

Para construir a *view* da página, vamos criar mais duas pastas dentro de ***views***: ***layouts*** e ***scripts***. Na pasta ***layouts*** criaremos um arquivo de *layout* padrão chamado **layout.phtml** com a seguinte configuração:

```
01  <html>
02  <head>
03    <meta http-equiv="Content-Type"
04      content="text/html; charset=ISO-8859-1" />
05    <title>Green Shopping</title>
06    <link rel="stylesheet" type="text/css"
07      media="screen" href="/default/css/estilo.css" />
08  </head>
09  <body>
10    <div id='conteudo'>
11      <?= $this->layout()->content ?>
12    </div>
13  </body>
14  </html>
```

Feito isso, vamos criar também a folha de estilos, a qual fizemos referência na *tag* ***link***, linhas 6 e 7. A folha de estilos é um arquivo público e justamente por isso deverá ficar dentro de ***www***. Mais especificamente, o

arquivo **estilos.css** que iremos criar deve ficar no seguinte diretório de nosso projeto:

ProjetoGreenShopping/www/default/css/estilos.css

O conteúdo de **estilos.css** será:

```
01  body {
02     background:url('/default/img/background.png') center;
03      background-position:top; height:842px;
04      font-family:Courier, monospace; font-weight:bold;
05      color:white; font-size:12px;
06  }
07
08  #conteudo{
09     width:858px; height:800px; margin:0 auto; margin-top:230px;
10  }
11
12  a#link          {
13     color:white; width:450px; height:35px; display:block;
14     background:url('/default/img/fundo_link_hover.png');
15     margin:10px auto; font-size:20px; padding-top:15px;
16     text-align:center; text-decoration:none;
17  }
18
19  a#link:hover {
20     background-position: 0 -100px;
21  }
22
23  #loja           {
24     background:url('/default/img/fundo_loja.png');
25     width:500px; margin:10px auto; clear:both; padding:10px;
26  }
27
28  #loja .nome     {
29     font-size:28px; display:block; background-color:#2EAF33;
30     text-align:center; padding:15px 0 15px 0; margin-bottom:10px;
31  }
32
33  #loja .label{
34     display:block;
35  }
36
37  #loja .value{
38     display:block;  margin-bottom:10px; font-family:fantasy;
39     font-size:20px;
40  }
41
42  #loja .frase{
43     display:block;  color:#2EAF33; margin-top:10px;
44     background-color:#FFF; text-align:center; font-family:sans-serif;
45     font-size:14px;     padding:10px 0 10px 0;
```

```
46  }
```

Não vem ao caso explicar detalhadamente a folha de estilos acima, portanto, vamos prosseguir com a criação dos demais arquivos da *view*. Para concluir a construção da *view*, criaremos agora dois arquivos de *scripts* que deverão estar dentro do seguinte diretório:

ProjetoGreenShopping/app/modules/default/views/scripts/index

Estes arquivos serão: **index.phtml** e **loja.phtml**. O conteúdo destes arquivo só fará sentido quando as suas respectivas *actions* estiverem implementadas. Por enquanto, nos limitaremos a apresentar o conteúdo destes arquivos, começando pelo arquivo **index.phtml**:

```
01  <? foreach($this->lojas as $loja) { ?>
02    <a id="link" href="/default/index/loja/id/<?= $loja->id ?>">
03      <span class="alt"><?= $loja->nome ?></span>
04    </a>
05  <? } ?>
```

E o arquivo **loja.phtml**:

```
01  <div id='loja'>
02
03    <span class="nome"><?= $this->loja->nome ?></span>
04
05    <span class="label">descrição:</span>
06    <span class="value"><?= $this->loja->descricao ?></span>
07
08    <span class="label">fone:</span>
09    <span class="value"><?= $this->loja->fone ?></span>
10
11    <span class="label">e-mail:</span>
12    <span class="value"><?= $this->loja->email ?></span>
13
14    <? foreach($this->loja->Mensagem as $mensagem) { ?>
15      <span class="frase"><?= $mensagem->texto ?></span>
16    <? } ?>
17
18  </div>
```

Todos os arquivos necessários para a *view* funcionar já foram criados. Precisamos apenas lembrar de ativar o recurso de *layout* do Zend Framework inserindo a seguinte linha no arquivo **module-init.php** do módulo **default**, uma vez que pretendemos utilizar o recurso de *layout* neste módulo:

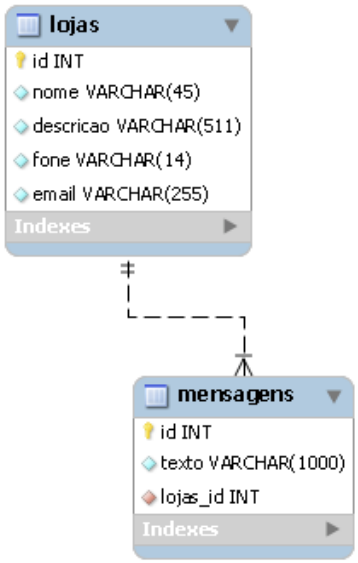
```
Zend_Layout::startMvc(dirname(__FILE__).'/views/layouts');
```

Com os arquivos da *view* criados e as configurações necessárias realizadas, podemos experimentar ir até o navegador e ver o que acontence com nossa página. O *layout* será carregado, mas devemos receber um *Warning* alegando que passamos um argumento inválido para um **foreach** em nosso arquivo de *script* **index.phtml**.

Isso aconteceu porque, nesse momento, ainda não foram carregadas na *view* as entidades necessárias para renderizar corretamente a página principal de nosso site. Resolveremos este problema mais adiante, uma vez que, para resolvê-lo, precisamos criar a base de dados e todos os arquivos necessários para que nossas duas entidades – **Loja** e **Mensagem** – possam trafegar através do canal de comunicação.

### *7.2.4 Criando a base de dados*

Necessitamos de uma base de dados onde seja possível armazenar nossas duas entidades e descrever o seu relacionamento. Descreveremos como deverá ficar a base de dados através de um Modelo Entidade-Relacional para o nosso problema:




Para materializar tal modelagem, criaremos uma base de dados MySQL chamada **base_greenshopping** através da execução do seguinte código SQL:

```
CREATE TABLE lojas (
  id int(11) NOT NULL auto_increment,
  nome varchar(45) default NULL,
  descricao varchar(511) default NULL,
  fone varchar(14) default NULL,
  email varchar(255) default NULL,
  PRIMARY KEY  (id)
) ENGINE=InnoDB;

CREATE TABLE mensagens (
  id int(11) NOT NULL auto_increment,
  texto varchar(100) default NULL,
  loja_id int(11) default NULL,
  PRIMARY KEY  (id),
  KEY fk_Mensagem_Loja (loja_id)
) ENGINE=InnoDB;
```

Feito isso, precisaremos configurar nosso *module* **default** para acessar esta base de dados, isso é feito modificando a cláusula **DATABASE** do seu arquivo **Module.ini**:

```
01  [DATABASE]
02  HOST              = localhost
03  NAME              = greenshopping
04  USER              = root
05  PASS              = root
06  PORT              =
07
08  [SEPTO_RESOURCES]
09  SeptoCyclo        = true
10  SeptoPanel        = true
```

Observe que, além de configurar o acesso à base de dados através da cláusula DATABASE, já aproveitamos para dizer que este módulo **default** deverá trabalhar com dois Septo Modules: o Septo Cyclo e o Septo Panel.

Nossa intenção é a de que, a partir desse ponto, possamos utilizar tais Septo Modules para continuar com o desenvolvimento de nossa aplicação. Para isso, além de constar no arquivo **Module.ini** do módulo **default** que tais recursos serão utilizados, temos que instalar o Septo Cyclo e o Septo Panel seguindo os passos apresentados no capítulo 9.

### *7.2.5 Criando o arquivo de configuração entities.yml*

Com o Septo Cyclo instalado e a base de dados criada, criaremos rapidamente o arquivo de configuração **entities.yml** que está faltando. Para isso, basta acessar o Septo Cyclo pelo navegador atráves do seguinte endereço:

http://localhost/SeptoCyclo

O Septo Cyclo irá exibir sua tela de abertura onde deveremos escolher o *module* com o qual pretendemos trabalhar e, nesta caso, será o **default**. Com o *module* **default** selecionado para basta clicar na seta que liga a imagem “DB TABLES” a imagem “YAML FILE”. Feito isso, uma imagem de confirmação irá aparecer no canto superior esquerdo da tela e o arquivo com o

mapeamento das entidades passará a estar presente no seguinte local dentro de nossa aplicação:

/ProjetoGreenShopping/app/modules/default/configs/entities.yml

Voltamos à IDE Eclipse para visualizar este arquivo e observamos que ele encontra-se da seguinte forma:

```
01 Lojas:
02   tableName: lojas
03   columns:
04     id:
05       type: integer(4)
06       primary: true
07       autoincrement: true
08     nome: string(45)
09     descricao: string(511)
10     fone: string(14)
11     email: string(255)
12 Mensagens:
13   tableName: mensagens
14   columns:
15     id:
16       type: integer(4)
17       primary: true
18       autoincrement: true
19     texto: string(100)
20     loja_id: integer(4)
21   relations:
22     Lojas:
23       local: loja_id
24       foreign: id
25       type: one
```

Faremos agora, antes de passar para a geração do código PHP que irá compor o canal de comunicação, algumas alterações em nossa arquivo de configuração. Vamos começar trocando o nome das classes que deverão ser criadas para o singular. Dessa forma, na linha X e Y, onde estava escrito Lojas e Mensagens, agora estará escrito Loja e Mensagem, respectivamente.

Outra alteração que faremos é adicionar um atributo relacional chamado mensagens em nossa entidade Loja que faz referência à todas as mensagens de uma determinada Loja. Com essa alteração será possível perguntar para um objeto do tipo Loja quais são os objetos do tipo Mensagem que ele possui. Da forma como estava, apenas o objeto Mensagem saberia dizer a qual objeto tipo Loja ele pertence.

Após a alteração do nome das classes para o singular e da adição de um atributo relacional em Loja, o arquivo **entities.yml** deve ficar da seguinte forma:

```
01 Loja:
02   tableName: lojas
03   columns:
04     id:
05       type: integer(4)
06       primary: true
07       autoincrement: true
08     nome: string(45)
09     descricao: string(511)
10     fone: string(14)
11     email: string(255)
12   relations:
13     Mensagem:
14       local: id
15       foreign: loja_id
16       type: many
17 Mensagem:
18   tableName: mensagens
19   columns:
20     id:
21       type: integer(4)
22       primary: true
23       autoincrement: true
24     texto: string(100)
25     loja_id: integer(4)
26   relations:
27     Loja:
28       local: loja_id
29       foreign: id
30       type: one
```

### *7.2.6 Criando o código PHP que compõe o canal de comunicação*

Uma vez gerado o arquivo **entities.yml** e feito as devidas alterações, agora é momento de gerar todas os arquivos necessários para lidar com as entidades **Loja** e **Mensagem** através do canal de comunicação.

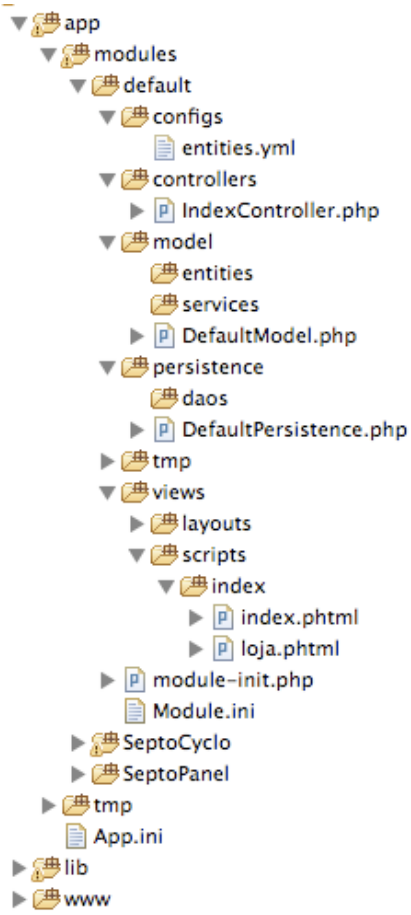
Para isso, basta navegar até o Septo Cyclo seguindo o endereço:

http://localhost/SeptoCyclo

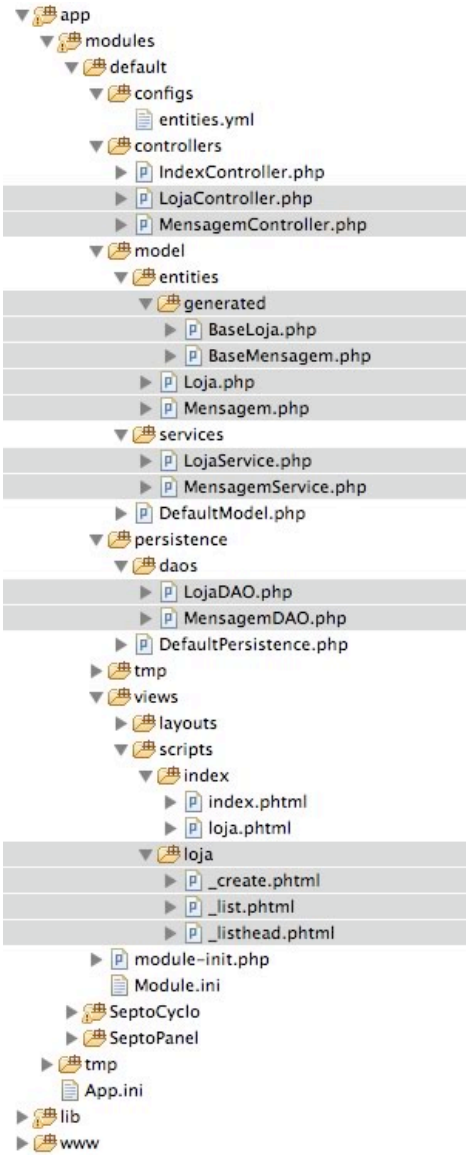
Um vez dentro dele, vamos selecionar o *module* **default** que é o que nos interessa. Com o *module* **default** selecionado, veremos a seguinte tela:


Para criar os arquivos que irão compor o canal de comunicação, basta clicar na seta que liga a imagem “YAML FILE” à imagem “PHP CODE”. Feito isso, uma mensagem deve aparecer no canto superior esquerdo da tela do Septo Cyclo informando que os arquivos necessários foram criados.

Para termos uma melhor noção de todos os arquivos que serão serão gerados com essa operação, abaixo temos toda a estrutura da aplicação antes de realizar esta operação:




E como ficou a aplicação depois de pedir para gerar todo o código necessário a partir do arquivo de configuração **entities.yml**:




### *7.2.7 Utilizando o SeptoPanel para cadastrar Lojas e Mensagens*

Se tudo correu bem, agora já podemos administrar as entidade **Loja** e **Mensagem** através do Septo Panel sem a necessidade de escrever nenhuma linha de código PHP ou HTML para estas entidades.

Vamos entrar no Septo Panel digitando o seguinte endereço:

http://localhost:8888/SeptoPanel

Deverá aparecer a tela de entrada do Septo Panel, onde é possível escolher com qual módulo queremos trabalhar. Escolhemos o *module* **default** e,

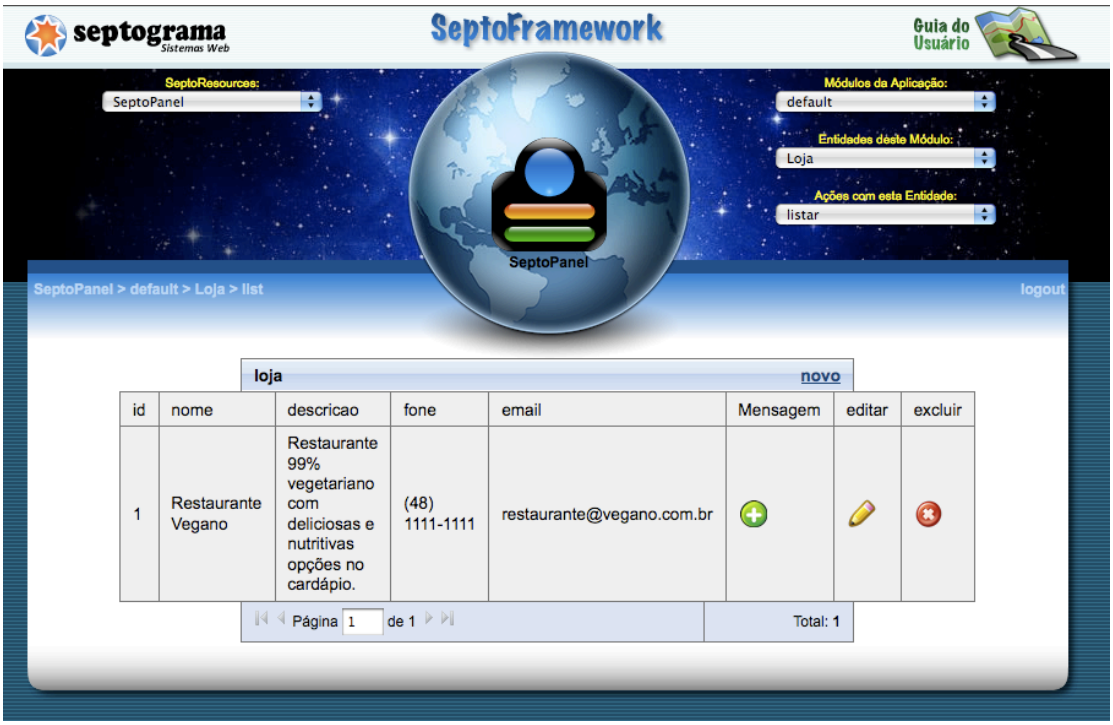
na tela seguinte, poderemos ver as duas entidades deste *module*: Loja e Mensagem:


Começaremos adicionando uma Loja e, a seguir, adicionaremos algumas mensagens a esta Loja. Para isso basta clicar em Loja e pedir para inserir um nova loja no canto superir direito da lista vazia de lojas que deve aparecer:


Uma vez dentro do formulário de Lojas, podemos preenchê-lo da seguinte forma:

| Campo | Valor |
|---|---|
| Nome: | Restaurante Vegano |
| Descrição: | Restaurante 100% vegetariano com deliciosas e nutritivas opções no cardápio. |
| Fone: | (99) 1111-1111 |
| Email: | restaurante@vegano.com.br |

Ao clicar no botão “salvar” uma mensagem de confirmação deverá aparecer. Ao pedirmos para listar novamente as entidades do tipo Loja, veremos a loja que acaba de ser inserida:




Podemos observar que, além dos botão para “editar”e “excluir”, também aparece um novo para “adicionar” mensagens para a loja que acabamos de inserir. Ao clicar neste botão serão listadas as mensagens da loja Restaurante Vegano. Como não existe nenhuma mensagem cadastrada nessa momento, a lista estará vazia. No canto direito superior da lista de mensagens que está vazia está o botão “novo”. Vamos utilizar este botão agora para inserir novas mensagens para esta loja.

Ao clicar no botão novo um formulário pedindo para escrever a nova mensagem para a loja Restaurante Vegano deve aparecer. Preencheremos o formulário com a mensagem “Suco grátis no sábado e no domingo!” e pedimos para salvar. Logo após inseriremos mais duas mensagens para esta loja: “Também fazemos entregas em domicílio.” e “Assistas palestras sobre alimentação natural nas sextas às 18h30.”

### *7.2.8 Criando as Actions da Página Principal*

Agora que temos nossa primeira loja devidamente cadastrada, podemos terminar de criar as páginas principais do site. Como os arquivos da *view* já foram criados, nos falta criar as *actions* que devem atender as solicitações do cliente.

Na sessão 7.2.2 começamos a criar o **IndexController** do módulo **default**, quem deverá atender as solicitações para a página principal. Abaixo, alteramos este *controller* para permitir que a navegação possa acontecer:

```
01  <?php
02
03  class IndexController extends Zend_Controller_Action {
04
05    public function indexAction() {
06      $model = Septo::getInstance()->getModel();
07       $service = $model->getService("loja");
08       $this->view->lojas = $service->getAll();
09    }
10
11    public function lojaAction() {
12      $id = $this->getRequest()->getParam("id");
13      $model = Septo::getInstance()->getModel();
14      $service = $model->getService("loja");
15      $this->view->loja = $service->getById($id);
16    }
17  }
```

Observe que o *controller* passou a ter duas *actions*. Cada uma delas servirá para cada uma das duas telas da aplicação. A primeira tela, a tela principal, está relacionada com a *action* **indexAction**, na linha 5 à 8.

Vamos analisar aqui cada uma das três linhas que compõem o corpo desta primeira *action*. Na linha 6 utilizamos a nossa instância de da classe Septo para solicitar que nos retorne um objeto que é a fachada do *module* **default**. Na linha 7, pedimos à fachada do *module* **default** que nos ofereça o um

serviço relacionado à entidade Loja. Na linha 8 pedimos para este serviço retornar todas as entidades lojas que encontrar na base de dados e colocamos este valor na *view* para que, uma vez dentro do arquivo de *script*, possamos utilizar estes valores para criar a página principal da aplicação.

O *script* que será utilizado para criar a página relacionada a esta primeira *action* será o arquivo **index.phtml**, apresentado na sessão 10.2.3. Neste *script* será criado um *link* para a nossa outra *action*, **lojaAction**, cuja função é criar uma outra página exibindo todos os detalhes de uma loja específica.

Em **lojaAction**, *action* que vai da linha 11 à 16, a primeira operação feita é pegar o *id* da loja que devemos exibir (linha 12). Com o identificador do objeto loja na mão, conseguimos um instância do serviço loja (linhas 13 e 14) e pedimos para este serviço que nos retorne a loja que possui o id informado utilizando o método ***getById*** (linha 15). Na mesma linha 15, atribuimos este valor a variável loja que passará a fazer parte da *view*, tornando-se acessível no contexto do arquivo de *script* e permitindo que nossa página contendo os detalhes da loja possa ser criada.

### *7.2.9 Navegação no site*

Entendido o processo de criação de novas lojas e mensagens, vamos agora inserir um conjunto completo de lojas e suas respectivas mensagens através do Septo Panel, conforme a tabela a seguir:

| **Lojas** | | |
|---|---|---|
| **Loja 1** | Nome: | *Restaurante Vegano* |
| | Descrição: | *Restaurante 100% vegetariano com deliciosas e nutritivas opções no cardápio.* |
| | Telefone: | *(99) 1111-1111* |
| | Email: | *restaurante@vegano.xyz.br* |
| | Mensagens: | *Suco grátis na segunda e na terça feira!* |
| | | *Também fazemos entregas em domicílio.* |
| | | *Palestras sobre alimentação natural toda sexta às 20 horas.* |
| **Loja 2** | Nome: | *Cine Popular* |
| | Descrição: | *Vá ao cinema e assista algo inteligente com seus amigos por apenas 2 reais!* |
| | Telefone: | *(99) 2222-2222* |

| | | |
|---|---|---|
| | Email: | *cinepopular@cinepopular.xyz.br* |
| | Mensagens: | *Em cartaz: Muito além do cidadão Kane* |
| | | *Em cartaz: Terráqueos* |
| | | *Em cartaz: Surplus* |
| **Loja 3** | Nome: | *Livraria Utopia* |
| | Descrição: | *Grande coleção de livros considerados grandes clássicos.* |
| | Telefone: | *(99) 3333-3333* |
| | Email: | *sac@utopialivraria.xyz.br* |
| | Mensagens: | *Compre 3 livros, ganhe 1!* |
| | | *Atendimento: segunda a sábado, das 8 às 20 horas.* |
| | | *Encontre aqui obras completas de Jung.* |
| **Loja 4** | Nome: | *Vista a Vida* |
| | Descrição: | *Vestuário produzido sem matar ou explorar qualquer animal.* |
| | Telefone: | *(99) 4444-4444* |
| | Email: | *atendimento@vistaavida.xyz.br* |
| | Mensagens: | *Chegaram novos modelos de camisa com temas veganos.* |
| | | *Descontos especiais para estudantes!* |
| | | *Atendimento de segunda a sexta, das 8 às 20 horas.* |

Feito isso, a primeira página do site deverá ficar da seguinte forma:

Ao escolher uma das lojas, deverá aparecer uma outra página com todos os detalhes e mensagens da loja:


## 7.3 Considerações Finais

Neste capítulo mostramos como criar uma aplicação utilizando o Septo Framework. Contudo, tal aplicação não estará completa sem uma implementação de um controle de acesso, daí a importância da conclusão do Septo Access, Septo Module apresentado no capítulo 9.

Outra consideração importante a ser feita é que não tramos de mostrar como funciona a validação dos valores informados através dos formulários, isso deve-se ao fato de que esta validação deve ter um comportamente diferenciado dependendo se estamos utilizando do Septo Panel ou não. Testes mais recentes mostram que o recurso de validação dos campos do

formulário deverá passar por uma reformulação para funcionar corretamente nos dois casos, daí termos optado por não apresentá-lo nessa oportunidade.

Em nossos projetos utilizando o Septo Framework temos notado que as maiores dificuldades costumam aparecer quando estamos lidando com problemas reais. Mas não é possível relatar aqui todos os possíveis problemas que podem surgir e suas respectivas soluções.

A nossa intenção em dedicar um capítulo inteiro para mostrar com criamos uma aplicação completa, embora bastante simples, é a de que isso possa servir como uma forma de compatilhar um pouco da experiência no desenvolvimento de sistemas utilizando o Septo Framework, mas permanecendo cientes de que as possibilidades e os problemas previstos aqui são pequenos diante dos desafios reais que poderão aparecer.

# 8. TRABALHOS FUTUROS

Em maio de 2008 tivemos o privilégio de participar, na cidade de Itajaí/SC, do PHPSC Conf 2009. Naquela oportunidade, conhecemos dois profissionais com perfis bem diferentes que somaram muito em conhecimento e inspiração para nós. Um deles foi Adler Medrado, primeiro e único profissional brasileiro com certificação Zend Framework, o outro foi Guilherme Blanco.

Guilherme Blanco é membro ativo do grupo que desenvolve e mantém o projeto Doctrine. Com uma apresentação descontraída e informal, passou a maior parte do tempo mostrando o que é Doctrine e o que ele é capaz de fazer.

Nenhuma surpresa durante a maior parte da sua apresentação. Até porque, nossa experiência com o Doctrine já havia nos levado há um estado onde conhecíamos muito dos seus pontos fortes e pontos fracos. Mas, pouco antes de acabar a palestra, Guilherme Blanco deu uma prévia em primeira mão daquilo que seria a versão 2.0 do Doctrine.

Dentre as novas características, o Doctrine 2 passaria a ser capaz de fazer o mapeamento das entidades utilizando notações logo acima dos seus atributos. Dessa forma, seria possível fazer com que as entidades deixassem de herdar **Doctrine_Record** e passassem a ser objetos simples, da forma como sempre gostaríamos que fosse.

Para completar, entraria em cena um novo personagem, o **Gerenciador de Entidades**, um objeto capaz de centralizar todo o poder do Doctrine e trabalhar com as novas entidades que agora trazem seus atributos mapeados via notações.

Esse conjunto pequeno e significativo de inovações eliminaria a barreira aparentemente intransponível que encontramos na versão atual do Doctrine que nos havia impedido de criar camadas independentes e abriu caminho para que o Septo Framework pudesse implementar o padrão *Tree layers architecture*.

Blanco prometeu o lançamento da versão de testes do novo Doctrine ainda para setembro de 2009 e a nossa vontade de ver aquela ferramenta pronta era tão grande que decidimos conter o ânimo e não acreditar muito na promessa.

Mesmo assim, conforme ia se aproximando o mês de setembro, as nossas visitas ao site do projeto Doctrine tornaram-se mais e mais frequentes e no dia 1 de setembro, superando todo o nosso ceticismo, a versão de testes do Doctrine 2.0 foi disponibilizada para download no site do projeto.

Dado a necessidade de apresentar a ferramenta Septo Framework ainda este ano, decidimos concluí-lo mantendo sua compatibilidade apenas com as versões anteriores do Doctrine. Mas, tão logo seja possível, todo o Septo Framework será reestruturado para trabalhar com o Doctrine 2 e, graças a isso, ganhar uma arquitetura ainda mais coerente, tranquila de manter e fácil de utilizar.

# 9. CONCLUSÕES

Além da própria ferramenta desenvolvida - o Septo Framework - a experiência somada ao longo desses dois anos em desenvolvimento de sistemas web utilizando PHP serviu como uma graduação a parte. O alimento principal para o desenvolvimento desta ferramenta foi a emoção de criar algo aplicável no mundo real capaz de tornar muito mais prático, coerente e fácil de manter os sistemas desenvolvidos utilizando tecnologia PHP.

Como não foi definido em detalhes desde o início do projeto até onde pretendíamos chegar, novas funcionalidades foram sendo agregadas ao Septo Framework sempre que possível. Hoje, o Septo Framework possuí inclusive recursos que se quer chegaram a ser citados durante todo o trabalho. Entre eles, está o **Septo Map**, cuja função, dentre outras coisas, é a de oferecer um resumo completo sobre todos os *modules* que compõem a aplicação e quais entidades esses *modules* possuem. Tal recurso levou semanas até começar a funcionar da maneira que esperávamos, mas valeu a pena porque tornou possível o trabalho de todos os **Septo Modules** uma vez que estes, para funcionar, precisam de informações precisas sobre tudo que contém a aplicação.

Várias vezes tomamos decisões de projetos pouco sábias que resultaram em muito esforço de desenvolvimento em vão. Houve momentos, por exemplo, onde decidimos desenvolver nosso próprio framework MVC e uma ferramenta ORM quando poderíamos utilizar, desde o início, o Zend Framework e o Doctrine, respectivamente.

Mas não fosse a permissão para errar, certamente não teríamos chegados a ter maturidade suficiente enquanto desenvolvedores para concluir com êxito tal ferramenta. A nossa esperança é a de que este trabalho possa servir como trampolim para todas as pessoas que buscam desenvolver sistemas PHP com qualidade e compreendam que pode ser interessante aproveitar da experiência que tivemos para começar com mais maturidade, evitar esforços desnecessários e ampliar a chance de sucesso de seus projetos.

# 10. REFERÊNCIAS

ALLEN, Rob; LO, Nick; BROWN, Steven. **Zend Framework in Action**. USA: Manning Publication, 2007.

DOCTRINE. **Guide to Doctrine for PHP**. USA, 2009. Disponível em: <http://www.doctrine-project.org/documentation/manual/1_1/en/pdf>. Acesso em: 29 set. 2009.

EVANS, Clarck C. **YAML**. México, 2009. Disponível em: <http://www.yaml.org/spec/1.2/spec.html>. Acesso em: 30 set. 2009.

FOWLER, Martin. **Patterns of Enterprise Application Architecture**. Chicago: Pearson, 2003.

FREEMAN, Eric; FREEMAN, Elisabeth. **Head First Design Patterns**. USA: Alta Books, 2007.

FREEMAN, Eric; FREEMAN, Elisabeth. **Head First HTML with CSS & XHTML**. 2. ed. USA: Alta Books, 2008.

HEJLSBERG, Anders. **Object-Relational Mappings**. USA, 2003. Disponível em: <http://www.artima.com/intv/abstract3.html>. Acesso em: 11 junho 2009.

MICROSOFT CORPORATION. **Implementing Interoperability Design Elements.** USA, 2003. Disponível em: <http://msdn.microsoft.com/en-us/library/ms998418.aspx>. Acesso em 07 out. 2009.

MYSQL AB, **MySQL Reference Manual**. USA: Cupertino, 2008. Disponível em: <http://dev.mysql.com/doc>. Acesso em: 20 ago. 2009.

PESSOA, Diego. **Desenvolvimento MVC com Zend Framework**. João Pessoa, 2008. Disponível em: <http://www.slideshare.net/felipernb/desenvolvimento-mvc-com-zend-framework-presentation>. Acesso em: 25 março 2009.

PHP. **Manual do PHP**. Brasil, 2009. Disponível em: <http://www.php.net/manual/pt_BR>. Acesso em: 28 ago. 2009.

SUN MICROSYSTEMS. **Core J2EE Patterns - Data Access Object**. USA: Santa Clara, 2007. Disponível em: <http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html>. Acesso em: 21 ago. 2009.

SUN MICROSYSTEMS. **Distributed Multitiered Applications**. USA: Santa Clara, 2007. Disponível em: <http://java.sun.com/javaee/5/docs/tutorial/doc/bnaay.html>. Acesso em: 17 set. 2009.

SUN MICROSYSTEMS. **Model-View-Controller**. USA: Santa Clara, 2002. Disponível em: <http://java.sun.com/blueprints/patterns/MVC-detailed.html>. Acesso em: 9 ago. 2009.

W3C, **HyperText Markup Language**. USA, 2008. Disponível em: <http://www.w3.org/html>. Acesso em: 11 julho 2009.

WOLFF, Markus. **Extending Zend_Controller_Action**. Hamburg, 2006. Disponível em: <http://blog.wolff-hamburg.de/archives/5-Extending-Zend_Controller_Action.html>. Acesso em: 3 junho 2009.

ZEND TECHNOLOGIES. **Zend Reference Guide**. USA: Cupertino, 2009. Disponível em: <http://framework.zend.com/manual/en>. Acesso em: 10 out. 2009.