UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ – UTFPR
CURSO SUPERIOR DE TECNOLOGIA EM DESENVOLVIMENTO DE SISTEMAS DE INFORMAÇÃO

FERNANDO FRANCISCO DE ANDRADE

DESENVOLVIMENTO DE APLICAÇÕES WEB COM A UTILIZAÇÃO DOS FRAMEWORKS CODEIGNITER E DOCTRINE

TRABALHO DE DIPLOMAÇÃO

MEDIANEIRA
2011

FERNANDO FRANCISCO DE ANDRADE

DESENVOLVIMENTO DE APLICAÇÕES WEB COM A UTILIZAÇÃO DOS FRAMEWORKS CODEIGNITER E DOCTRINE

Trabalho de Diplomação apresentado à disciplina de Trabalho de Diplomação, do Curso Superior de Tecnologia em Desenvolvimento de Sistemas de Informação – CSTADS – da Universidade Tecnológica Federal do Paraná – UTFPR, como requisito parcial para obtenção do título de Tecnólogo.

Orientador: M.Sc. Fernando Schütz.

MEDIANEIRA

2011

Ministério da Educação
Universidade Tecnológica Federal do Paraná
Diretoria de Graduação e Educação Profissional
Coordenação do Curso Superior de Tecnologia em
Desenvolvimento de Sistemas de Informação

TERMO DE APROVAÇÃO

DESENVOLVIMENTO DE APLICAÇÕES WEB COM A UTILIZAÇÃO DOS FRAMEWORKS CODEIGNITER E DOCTRINE.

Por

FERNANDO FRANCISCO DE ANDRADE

Este Trabalho de Diplomação (TD) foi apresentado às 14:00 h do dia 14 de junho de 2011 como requisito parcial para a obtenção do título de Tecnólogo no Curso Superior de Tecnologia em Desenvolvimento de Sistemas de Informação, da Universidade Tecnológica Federal do Paraná, *Campus* Medianeira. O candidato foi argüido pela Banca Examinadora composta pelos professores abaixo assinados. Após deliberação, a Banca Examinadora considerou o trabalho aprovado.

Prof. Fernando Schütz UTFPR – *Campus* Medianeira (Orientador)	Prof. Itamar Pena Nieradka UTFPR – *Campus* Medianeira (Convidado)
Prof. Márcio Angelo Matté UTFPR – *Campus* Medianeira (Convidado)	Prof. Juliano Rodrigo Lamb UTFPR – *Campus* Medianeira (Responsável pelas atividades de TCC)

A folha de aprovação assinada encontra-se na Coordenação do Curso.

AGRADECIMENTOS

Agradeço a minha família e amigos pelo incentivo, apoio e confiança depositados nesta formação. Em especial agradeço minha mãe Maria Francisca de Andrade, meu pai Fernandes Pinto de Andrade.

Agradeço ao meu colega de trabalho André Marcelo Weiss que sempre pude contar quando precisei.

A todos os professores que dedicaram atenção e apoio, em especial aos professores Everton Coimbra de Araújo e Fernando Schütz pelas orientações em estágio e trabalho de conclusão de curso respectivamente.

Aos colegas de universidade, pois juntos dividimos as mais diversas atividades durante este período, e que dentre estes alguns se tornaram grandes amigos.

RESUMO

ANDRADE, Fernando Francisco. Desenvolvimento de aplicações *WEB* com a utilização dos *Frameworks CodeIgniter e Doctrine*. 2011. Trabalho de Conclusão de Curso, Universidade Tecnológica Federal do Paraná. Medianeira, 2011.
O presente trabalho é um estudo sobre os dois *frameworks* para desenvolvimento de aplicações *WEB* em PHP, o *CodeIgniter* e *Doctrine*. São apresentadas características de cada um bem como funcionalidades e configurações necessárias para o uso. Ao final é apresentado um estudo de caso onde os *Frameworks* trabalham em conjunto para desempenhar as funcionalidades de uma aplicação *WEB*.

Palavras-chave: PHP, desenvolvimento *WEB*, *CodeIgniter*, *Doctrine*, MVC.

RESUMO EM LINGUA ESTRANGEIRA

ANDRADE, Fernando Francisco. Developing WEB applications using frameworks CodeIgniter and Doctrine. 2011. Work of Conclusion of Course, Federal Technological University of Paraná. Medianeira, 2011.
This is a study on the two Frameworks for developing WEB applications in PHP, CodeIgniter and Doctrine. Some characteristics of each as well as features and settings required to use. At the end is present a case study where the frameworks work together to perform the functionality of a WEB application.

Keywords: *PHP, WEB development, CodeIgniter, Doctrine, MVC.*

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE SIGLAS

API	*Application Programming Interface*
CMS	*Content Management System*
DQL	*Doctrine Query Language*
FTP	*File Transfer Protocol*
GNU	*GNU'a Not Unix*
GPL	*General Public Licence*
HQL	*Hibernate Query Language*
HTML	*HyperText Markup Language*
HTTP	*HyperText Transfer Protocol*
JPQL	*Persistence Query Language*
MVC	*Model-View-Controller*
ORM	*Object Relational Mapping*
RDBMS	*Relation Database Management System*
UI	*User Interface*
XHTML	*eXtensible HyperText Markup Language*
XML	*eXtensible Markup Language*
YAML	*Yet Another Markup Language*

SUMÁRIO

1 INTRODUÇÃO

Um *framework* é uma base de onde se pode desenvolver algo maior ou mais específico. É uma coleção de códigos-fontes, classes, funções, técnicas e metodologias que facilitam o desenvolvimento de *softwares*(MINETTO, 2007).

Com o grande avanço no desenvolvimento de aplicações *WEB*, é cada vez mais necessário desenvolver aplicações robustas de forma organizada e em tempo minimizado. Isso leva os programadores a deixarem de desenvolver suas aplicações totalmente do zero e passarem a utilizar cada vez mais ferramentas que auxiliam na construção de suas aplicações.

O desenvolvimento de aplicações *WEB* utilizando *frameworks* se expandiu de tal forma que se tornou imprescindível o uso de *frameworks* para construção de sistemas voltados para *internet/intranet/web*(SOUZA, 2010).

Os chamados *frameworks* facilitam a vida do programador ao possibilitar a criação de aplicações de forma mais rápida e eficiente, utilizando muitas vezes o padrão de projeto MVC (*Model View Controller*).

O padrão MVC encaixa-se como uma luva para aplicações *WEB*, e é melhor aproveitado quando explorado por um *frameworks*(SOUZA, 2010).

Os *frameworks* também são utilizados na chamada persistência de dados, que através de métodos dinâmicos de acesso a diversas bases de dados retornam os resultados em forma de objetos, tornando a aplicação de persistência de dados orientada a objetos, dando assim maior facilidade na manipulação das informações.

1.1 OBJETIVO GERAL

O objetivo deste trabalho é demonstrar através de um estudo e desenvolvimento de uma aplicação, o uso dos *frameworks CodeIgniter* para desenvolvimento de aplicações e *Doctrine* para mapeamento objeto-relacional e também a integração entre os mesmos.

1.2 OBJETIVOS ESPECÍFICOS

- Realizar uma consulta em livros, artigos e trabalhos escritos sobre os *frameworks CodeIgniter* e *Doctrine*;
- Desenvolver um estudo quanto à viabilidade do uso de *frameworks* na construção de aplicações;
- Analisar, projetar e desenvolver uma aplicação que servirá como estudo experimental para a demonstração de uso dos *frameworks* citados;

1.3 JUSTIFICATIVA

Um dos grandes problemas para o programador que inicia o desenvolvimento de uma nova aplicação é de como estruturá-lo de forma que ao longo da construção do *software*, ele não se torne um emaranhado de código de difícil compreensão, trazendo assim muita dificuldade na hora de fazer a manutenção do mesmo.

Os *frameworks* vêm com a proposta de auxiliar neste processo, já que trazem ao desenvolvedor um *core* de estrutura iniciado, que nada mais é do que uma estrutura base onde se trabalhada seguindo o seu padrão.

O *CodeIgniter* é um *frameworks* de desenvolvimento de aplicações para quem constrói sites em PHP. Seu objetivo, através de um conjunto de bibliotecas possibilita que se desenvolvam projetos mais rapidamente do que se estivesse codificando do zero. O *CodeIgniter* permite que se mantenha o foco em seu projeto minimizando a quantidade de código necessário para uma dada tarefa. Sua estrutura usa o padrão MVC, cujo objetivo primário é separar a modelagem de dados, da camada de apresentação de um aplicativo de software.

O *Doctrine* é um *frameworks* de mapeamento objeto-relacional (ORM) para PHP a partir da versão 5.2.3. Um de seus principais recursos é a opção de se escrever consultas de bancos de dados em um dialeto SQL proprietário orientado a objetos chamado *Doctrine Query Language* (DQL), que foi inspirada na *Hibernate Query Language* (HQL) do *framework Hibernate*[1] para *Java*. Isso fornece aos desenvolvedores uma alternativa poderosa ao SQL que mantém a flexibilidade sem duplicações desnecessárias de código. Ele utiliza o

[1] *Hibernate* é um *framework* para o mapeamento objeto-relacional escrito na linguagem *Java*, mas também é disponível em *.Net* como o nome *NHibernate*.

(*Design Pattern Active Record*) para fazer abstração entre as tabelas do banco de dados e as suas respectivas classes, ou seja, para cada tabela no banco teremos uma classe correspondente. Sendo possível delegar a responsabilidade da criação das classes para o *Doctrine*.

1.4 ESTRUTURA DO TRABALHO

Esse trabalho é dividido em quatro capítulos os quais são resumidamente descritos a seguir:

No primeiro capitulo é apresentada a introdução sobre o assunto abordado com os objetivos gerais e específicos, e uma justificativa sobre o tema.

No segundo capítulo é apresentado o estudo sobre algumas das tecnologias envolvidas e os *frameworks Doctrine* e *CodeIgniter*. São detalhadas suas características e descritas algumas de suas funcionalidades.

O terceiro capítulo apresenta o objetivo principal deste estudo, que é a implementação de um estudo experimental para demonstração dos *frameowrks*.

As considerações finais são apresentadas no quarto capítulo onde é apresentada uma conclusão para o trabalho e sugestões para implementações futuras.

2 REVISÃO BIBLIOGRÁFICA

De acordo com os objetivos deste trabalho, neste capítulo são apresentados os temas a serem contextualizados, antes do estudo dos *frameworks* propriamente ditos.

2.1 APLICAÇÕES WEB

Aplicações *WEB* são sistemas computacionais instalados e executados em servidores que, podem ou não, serem dispostos na Internet. Uma das vantagens em se desenvolver uma aplicação *WEB* é que elas não necessitam de instalação nos clientes. Isso facilita na atualização e manutenção do código fonte, já que o mesmo fica em um único lugar, e as atualizações ficam disponíveis simultaneamente para todos os usuários. Também vale citar o fato de consumir menos recurso de processamento dos clientes uma vez que, o mais "pesado" não é executado nos clientes.

Aplicações *WEB* consistem basicamente de:

- Navegador cliente que fornece a interface com o usuário (UI, do inglês "*User Interface*");
- Uso de alguma linguagem interpretada pelo navegador cliente para criação das telas (HTML, XHTML ou XML/XSL);
- Uso de HTTP ou HTTPS como protocolo de transmissão dos dados;
- Lógica de negócio (roda no servidor).

Na Figura 1 é demonstrado de forma resumida, como o cliente faz uma requisição *WEB* ao servidor solicitando o serviço de uma aplicação hospedada no mesmo.

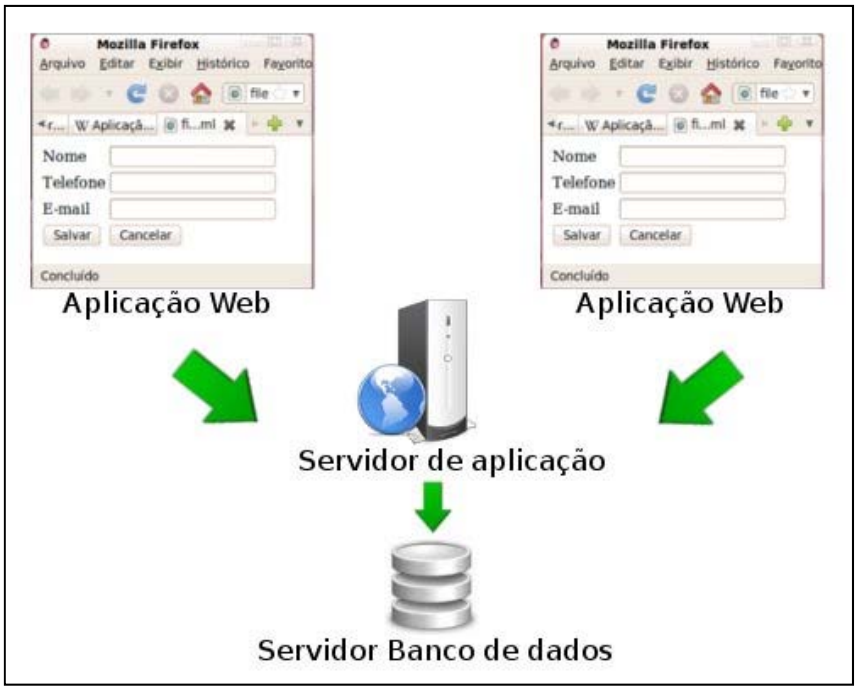

Figura 1 - Fluxo de uma requisição WEB
Fonte: Brunetta, 2011.

2.2 PHP

O PHP atualmente na versão 5.3.6 é uma linguagem de criação de *scripts* do lado do servidor que foi projetada especificamente para *WEB*. Dentro de uma página HTML, você pode embutir código de PHP que será executado toda vez que a pagina for visitada. O código de PHP é interpretado no servidor *WEB* e gera HTML ou outra saída que o visitante verá (Thomson, 2005).

O PHP foi concebido em 1994 como resultado do trabalho de uma única pessoa, *Rasmus Lerdorf*. O PHP foi adotado por outras pessoas inteligentes e foi reescrito três vezes para proporcionar o amplo e aperfeiçoado produto que vemos hoje. Em outubro de 2002, ele era utilizado em mais de nove milhões de domínios em todo o mundo e esse número está crescendo rapidamente. Você pode contatar o número atual em *http://www.php.net/usage.php* (Thomson, 2005).

O PHP significa originalmente *Personal Home Page*, mas foi alterado de acordo com a conversão para atributos de nomes recursiva do GNU (*GNU = GNU'a Not Unix*) e agora significa PHP *Hypertext Preprocessor* (Thomson, 2005).

A principal versão do PHP é a versão 5. Nesta versão, o *Zend Engine* foi completamente reescrito, e foram feitos aprimoramentos à linguagem (Thomson, 2005).

2.3 SERVIDOR WEB – APACHE

O servidor Apache (*Apache Server*) foi criado em 1994, por *Rob McCool* funcionário da NCSA e atualmente, é o servidor *WEB* mais bem sucedido, devido também ao fato de ser livre. Suas principais características são:

- Compatibilidade com a versão 1.1 do protocolo HTTP;
- Funcionalidades mantidas através de módulos que podem ser acoplados ao núcleo do *Apache*;
- Possibilidade de criar módulos baseados na API do servidor;
- Disponibilidade para plataformas de sistemas operacionais *Windows*, *Novell NetWare*, OS/2 e vários derivados *POSIX*, como, *Linux*, *UNIX*, *FreeBSD* entre outros;
- Módulos nativos para integração com linguagens de programação como *Java* (JSP), *PERL*, ou então páginas dinâmicas como PHP;
- Módulos de acesso a bancos de dados como *MySQL*;
- Configuração simples através de arquivos de texto puro (*httpd.conf*).

2.4 MYSQL

O *MySQL* atualmente na versão 5.5 é um sistema de gerenciamento de banco de dados relacional (*relation database management system* - RDBMS). Um banco de dados permite armazenar, pesquisar, classificar e recuperar dados de forma eficiente. O servidor de *MySQL* controla o acesso aos dados para assegurar que vários usuários possam trabalhar com os dados ao mesmo tempo, fornecer acesso rápido aos dados e assegurar que somente usuários autorizados obtenham acesso. Portanto, o *MySQL* é um servidor multiusuário e multiencadeado (ou *multitbreaded*). Ele utiliza SQL(*Structured Query Language*), a linguagem de consulta padrão de banco de dados em todo o mundo. O *MySQL* está publicamente disponível desde 1996, mas tem uma história de desenvolvimento que remonta a 1979. O *MySQL* ganhou o prêmio *Journal Readers' Choice Award Linux* em várias ocasiões (Thomson, 2005).

MySQL está disponível sob um esquema de licença dupla. Você pode usá-lo sob a licença *Open Source* (*GPL – General Public Licence*) gratuitamente, contanto que cumpra os termos da licença. No entanto, se quiser distribuir uma aplicação não-GPL que inclua o *MySQL*, você pode comprar uma licença comercial (Thomson, 2005).

2.5 MVC (MODEL - VIEW - CONTROLLER)

O Modelo é o objeto de aplicação, a Visão é a apresentação na tela e o Controlador é o que define a maneira como a interface do usuário reage às entradas do mesmo (GAMMA, 2000)

O padrão de projeto MVC (Modelo Visualização Controle) fornece uma maneira de dividir a funcionalidade envolvida na manutenção e apresentação dos dados de uma aplicação. O MVC não é novo e foi originalmente desenvolvido para mapear as tarefas tradicionais de entrada, processamento e saída para o Modelo de interação com o usuário. Usando o padrão MVC fica fácil mapear esses conceitos no domínio de aplicações *WEB* multicamadas (MACORATTI, 2002).

Na arquitetura do MVC o Modelo representa os dados da aplicação e as regras do negócio que governam o acesso e a modificação dos dados. O Modelo mantém o estado persistente do negócio e fornece ao controlador a capacidade de acessar as funcionalidades da aplicação encapsuladas pelo próprio Modelo (MACORATTI, 2002).

Um componente de visualização renderiza o conteúdo de uma parte particular do Modelo e encaminha para o controlador as ações do usuário e, acessa também os dados do Modelo via controlador e define como esses dados devem ser apresentados (MACORATTI, 2002).

Um controlador define o comportamento da aplicação, é ele que interpreta as ações do usuário e as mapeia para chamadas do Modelo. Em um cliente de aplicações *WEB* essas ações do usuário poderiam ser cliques de botões ou seleções de menus. As ações realizadas pelo Modelo incluem ativar processos de negócio ou alterar o estado do Modelo. Com base na ação do usuário e no resultado do processamento do Modelo, o controlador seleciona uma visualização a ser exibida como parte da resposta a solicitação do usuário. Há normalmente um controlador para cada conjunto de funcionalidades relacionadas (MACORATTI, 2002).

2.6 DOCTRINE

Doctrine é um mapeador objeto-relacional (ORM) para PHP 5.3.0 + que fornece persistência transparente para objetos do PHP. Ela fica em cima de uma poderosa camada de abstração de banco de dados (DBAL). *Object-Relational Mappers* tem a principal tarefa de fazer a conversão entre objetos (PHP) em linhas de dados relacional.

Uma das suas principais características é a opção de escrever consultas num dialeto SQL, chamado *Doctrine Query Language* (DQL), inspirado pelo dialeto do *Hibernate*, o HQL. Apesar de haver pequenas diferenças entre DQLs e as SQL normais, a abstração dos mapeamentos entre linhas de dados e objetos são consideráveis, permitindo aos desenvolvedores criar consultas poderosas de forma simples e flexível.

Como o termo ORM já sugere, o *Doctrine* visa simplificar a conversão de linhas de dados para o Modelo de objetos do PHP. A principal utilização para o *Doutrine* é, portanto, em aplicativos que utilizam a Programação Orientada a Objetos.

Doctrine está dividido em três pacotes principais:

- **Common** - O pacote *Common* contém componentes altamente reutilizáveis, que não têm dependências além do pacote em si (e PHP, é claro). O *namespace* raiz do pacote é *Common Doctrine\Common*.
- **DBAL** - O pacote DBAL contém uma camada de abstração de banco de dados melhorada sobre a do PDO, mas não está totalmente vinculada ao PDO. O propósito desta camada é fornecer uma API simples que faça pontes diferentes entre diferentes fornecedores de RDBMS. O *namespace* raiz do pacote é DBAL *Doctrine\DBAL*.
- **ORM** - O pacote contém o kit de ferramentas ORM mapeamento objeto-relacional que oferece persistência transparente para simples objetos relacionais PHP. O *namespace* raiz do pacote ORM é *Doctrine\ORM*.

2.6.1 Bootstrapping

Bootstrapping é um procedimento relativamente simples que é feito em apenas duas etapas:

- Garantir o carregamento por demanda dos arquivos das classes do *Doctrine*.
- Obter uma instância *EntityManager*.

2.6.2 Class Loading

Em primeiro lugar a classe de instalação deve ser carregada. Após isso é preciso configurar alguns carregadores de classes (muitas vezes chamado de "*autoloader*") para que os arquivos de classe do *Doutrine* sejam carregados sob demanda. O *namespace Doutrine* contém um carregador de classes muito rápido e minimizado que pode ser usado pelo *Doutrine* e quaisquer outras bibliotecas onde os padrões de codificação asseguram que a localização de uma classe na árvore de diretórios é refletida pelo seu nome e *namespace*.

O uso do carregador de classe do *Doutrine* não é obrigatório. O *Doutrine* não considera como as classes são carregadas, portanto um carregador de classes diferente pode ser utilizado.

O exemplo do Quadro 1 mostra a configuração de um *ClassLoader* para os diferentes tipos de instalações do *Doutrine*:

```php
<?php
require '/path/to/libraries/Doctrine/Common/ClassLoader.php';
$classLoader = new \Doctrine\Common\ClassLoader('Doctrine', '/path/to/libraries');
$classLoader->register();
```

Quadro 1 - Configuração do ClassLoader
Fonte: Doctrine Documentation

Depois de feito o carregamento da classe, deve ser criada uma instância *EntityManager*. A classe *EntityManager* é o primeiro ponto de acesso à funcionalidade fornecida pelo ORM *Doctrine*.

Como pode ser visto no Quadro 2, uma configuração simples do *EntityManager* requer uma instância de *Doctrine\ORM\Configuration*, bem como alguns parâmetros de conexão com banco de dados:

```php
<?php
use Doctrine\ORM\EntityManager,
    Doctrine\ORM\Configuration;
// ...
if ($applicationMode == "development") {
    $cache = new \Doctrine\Common\Cache\ArrayCache;
} else {
    $cache = new \Doctrine\Common\Cache\ApcCache;
}
$config = new Configuration;
$config->setMetadataCacheImpl($cache);
$driverImpl = $config->newDefaultAnnotationDriver('/path/to/lib/MyProject/Entities');
$config->setMetadataDriverImpl($driverImpl);
$config->setQueryCacheImpl($cache);
$config->setProxyDir('/path/to/myproject/lib/MyProject/Proxies');
$config->setProxyNamespace('MyProject\Proxies');
if ($applicationMode == "development") {
    $config->setAutoGenerateProxyClasses(true);
} else {
    $config->setAutoGenerateProxyClasses(false);
}
$connectionOptions = array(
    'driver' => 'pdo_sqlite',
    'path' => 'database.sqlite'
);
$em = EntityManager::create($connectionOptions, $config);
```

Quadro 2 - Configuração do ClassLoader
Fonte: Doctrine Documentation

2.6.3 Opções de Configuração

As seções seguintes descrevem todas as opções de configuração disponíveis em uma instância de *Doctrine\ORM\Configuration*.

- **Proxy Directory (* obrigatório*)** - Obtém ou define o diretório onde *Doctrine* gera as classes *proxy*.
- **Proxy Namespace (* obrigatório*)** - Obtém ou define o *namespace* a ser usado para as classes *proxy* geradas.
- **Metadata Driver (* obrigatório*)** - Obtém ou define a implementação do driver de metadados que é utilizada pelo *Doutrine* para adquirir os metadados objeto-relacionais para suas classes.
- **Metadata Cache (*recomendado*)** - Obtém ou define a implementação de *cache* para usar o *cache* de informações de metadados, isto é, todas as informações que se fornecer através de anotações, *xml* ou *yaml*, de modo que

não precisa ser analisado e carregado a partir do zero a cada requisição, que é um desperdício de recursos. A implementação do *cache* deve implementar a interface *Doctrine\Common\Cache*.

- **Query Cache (*recomendado*)** - Obtém ou define a implementação de cache a ser usado para consultas de *cache* DQL, ou seja, o resultado de um processo de análise DQL que inclui o SQL final, bem como informações sobre como metadados para processar o conjunto de resultados de uma consulta SQL.
- **SQL Logger (*opcional*)** - Obtém ou define o *logger* a ser usado para registrar todas as instruções SQL executadas pelo *Doctrine*. A classe *logger* deve implementar a interface *Doctrine\DBAL\Logging\SQLLogger*.
- **Auto-generating Proxy Classes (*opcional*)** - Obtém ou define se as classes de *proxy* deve ser gerada automaticamente em tempo de execução pelo *Doutrine*. Se definido como *FALSE*, as classes de *proxy* devem ser geradas manualmente através do *Doctrine* por linha de comando.

2.6.4 Docblock Annotations

Docblock Annotations é uma ferramenta para incorporar *metadados* dentro da seção de documentação que pode ser processado por outra ferramenta. O *Doutrine* generaliza o conceito de *Annotations* para que possam ser usadas para qualquer tipo de *metadados* e de modo que seja fácil de definir novas *Annotations*.

A fim de permitir que os valores de anotação mais envolvidos e para reduzir as possibilidades de confrontos com outras *Annotations*, o *Doutrine* apresenta uma sintaxe alternativa para as *Annotations* que é fortemente inspirada na sintaxe introduzida no Java 5.

Para que uma classe seja marcada como um objeto-relacional persistente ela deve ser definida como entidade. Isso pode ser feito através da *Annotation @Entity* como mostrado no Quadro 3.

```php
<?php
/** @Entity */
class MyPersistentClass
{
    //...
}
```

Quadro 3 - Annotation @Entity

Fonte: Doctrine Documentation

Por padrão, a entidade será persistente à uma tabela com o mesmo nome que o nome da classe. Para mudar isso, é possível usar a anotação *@Table* como mostrado no Quadro 4:

```php
<?php
/**
 * @Entity
 * @Table(name="my_persistent_class")
 */
class MyPersistentClass
{
    //...
}
```

Quadro 4 - Annotation @Table
Fonte: Doctrine Documentation

2.6.5 Doctrine Mapping Type

O *Doctrine Mapping Type* define o mapeamento entre um tipo do PHP e um tipo SQL. Todos os tipos de mapeamento que acompanham *Doctrine* são totalmente portáveis entre diferentes SGBD. Abaixo uma visão rápida dos tipos de mapeamento do *Doctrine*:

- **string** - Tipo que mapeia um *VARCHAR* SQL para uma *string* do PHP.
- **integer** - Tipo que mapeia um *INT* SQL para um PHP *integer*.
- **Smallint** - Tipo que mapeia um *SMALLINT* de banco de dados para um PHP *integer*.
- **bigint** - Tipo que mapeia um *BIGINT* de banco de dados para uma string PHP.
- **boolean** - Tipo que mapeia um SQL *boolean* para um PHP *boolean*
- **decimal** - Tipo que mapeia um SQL *DECIMAL* para um PHP *double*
- **date** - Tipo que mapeia um SQL *DATETIME* para um objeto PHP *DateTime*.
- **time** - Tipo que mapeia um SQL *TIME* para um objeto PHP *DateTime*.

Depois de uma classe ter sido marcada como uma entidade que pode especificar mapeamentos para seus atributos. Para marcar uma propriedade de persistência relacional é usada a anotação *@Column*. O uso desta anotação pode ser vista no Quadro 5.

```php
<?php
/** @Entity */
class MyPersistentClass
{
    /** @Column(type="integer") */
    private $id;
    /** @Column(length=50) */
    private $name; // type defaults to string
    //...
}
```

Quadro 5 - Annotation @Column
Fonte: Doctrine Documentation

A *Annotation @Column* contem os seguintes atributos:

- **type**: (opcional) O tipo de mapeamento para usar para a coluna.
- **name**: (opcional) O nome da coluna no banco de dados.
- **length**: (opcional) O comprimento da coluna no banco de dados.
- **unique**: (opcional) Se a coluna é uma chave única.
- **nullable**: (opcional) Se a coluna de banco de dados pode ser nula.

2.6.6 Association Mapping

Ao mapear as associações bidirecionais, é importante entender o conceito de propriedade e os lados inversos. As seguintes regras gerais se aplicam:

- As relações podem ser bidirecionais ou unidirecionais.
- Uma relação bidirecional possui um lado e um lado inverso.
- Uma relação unidirecional apenas tem um lado proprietário.
- O lado proprietária de um relacionamento determina as atualizações para o relacionamento no banco de dados.

O lado inverso de uma relação bidirecional deve referir-se ao seu lado proprietário pelo uso do atributo *mappedBy* das declaração de mapeamento *OneToOne*, *OneToMany*, ou *ManyToMany*. O atributo *mappedBy* designa o campo da entidade que é o dono da relação.

2.6.7 Mapeamento de superclasse

Uma superclasse mapeada é uma classe abstrata ou concreta que fornece o estado de persistência de uma entidade e mapeamento de informações para suas subclasses, mais que em si não é uma entidade. O Quadro 6 mostra como este mapeamento é feito através de anotações.

```php
<?php
/** @MappedSuperclass */
class MappedSuperclassBase
{
    /** @Column(type="integer") */
    private $mapped1;
    /** @Column(type="string") */
    private $mapped2;
    /**
     * @OneToOne(targetEntity="MappedSuperclassRelated1")
     * @JoinColumn(name="related1_id", referencedColumnName="id")
     */
    private $mappedRelated1;

    // ... more fields and methods
}

/** @Entity */
class EntitySubClass extends MappedSuperclassBase
{
    /** @Id @Column(type="integer") */
    private $id;
    /** @Column(type="string") */
    private $name;

    // ... more fields and methods
}
```

Quadro 6 - Mapeamento de Superclasse
Fonte: Doctrine Documentation

2.6.8 Herança Tabela Única

É uma estratégia de mapeamento de herança, onde todas as classes da hierarquia são mapeadas para uma única tabela do banco de dados. O Quadro 7 mostra como utilizar este recurso.

```php
<?php
namespace MyProject\Model;

/**
 * @Entity
 * @InheritanceType("SINGLE_TABLE")
 * @DiscriminatorColumn(name="discr", type="string")
 * @DiscriminatorMap({"person" = "Person", "employee" = "Employee"})
 */
class Person
{
    // ...
}

/**
 * @Entity
 */
class Employee extends Person
{
    // ...
}
```

Quadro 7 - Herança tabela única
Fonte: Doctrine Documentation

2.6.9 Herança Class Table

É uma estratégia de mapeamento de herança, onde cada classe em uma hierarquia é mapeada para várias tabelas: a sua própria tabela e as tabelas de todas as classes pai. A tabela de uma classe filho é ligada à tabela de uma classe pai por meio de uma restrição de chave estrangeira. Como pode ser visto no Quadro 8.

```php
<?php
namespace MyProject\Model;

/**
 * @Entity
 * @InheritanceType("JOINED")
 * @DiscriminatorColumn(name="discr", type="string")
 * @DiscriminatorMap({"person" = "Person", "employee" = "Employee"})
 */
class Person
{
    // ...
}

/** @Entity */
class Employee extends Person
{
    // ...
}
```

Quadro 8 - Herança class table
Fonte: Doctrine Documentation

2.6.10 Entidade e mapa de identificações

Entidades são objetos com identificação. Sua identificação tem um significado conceitual dentro do seu domínio. Em um aplicativo CMS cada artigo tem um único ID. Você pode identificar cada artigo por esse id.

Veja o seguinte exemplo no Quadro 9, onde você encontra um artigo com o título "*Hello World*" com o ID 1234:

```php
<?php
$article = $entityManager->find('CMS\Article', 1234);
$article->setHeadline('Hello World!');

$article2 = $entityManager->find('CMS\Article', 1234);
echo $article2->getHeadline();
```

Quadro 9 - Entidades
Fonte: Doctrine Documentation

O *Doctrine* mantém um mapa de cada entidade e suas identificações que foram recuperadas por solicitação PHP e continua voltando-lhe as mesmas instâncias.

No exemplo anterior, o *echo* imprime "*Hello World*!" na tela. Para verificar que *$article* e *$article2* estão de fato apontando para a mesma instância basta executar o código mostrado no Quadro 10:

```php
<?php
if ($article === $article2) {
    echo "Yes we are the same!";
}
```

Quadro 10 - Mapa de identificação
Fonte: Doctrine Documentation

2.6.11 Persistindo Entidades

Uma entidade pode ser persistida, utilizando o método *EntityManager#persist($entity).* Ao aplicar a operação de persistência em alguma entidade, essa entidade se torna gerenciável, o que significa que a sua persistência a partir de agora gerida por um *EntityManager*. O Quadro 11 mostra um exemplo de uso do método *persist().*

```php
<?php
$user = new User;
$user->setName('Mr.Right');
$em->persist($user);
$em->flush();
```

Quadro 11 - Função persit
Fonte: Doctrine Documentation

2.6.12 Removendo Entidades

Um entidade pode ser removida do armazenamento persistente através do método *EntityManager#remove($entity)*, mostrado no Quadro 12.

```php
<?php
$em->remove($user);
$em->flush();
```

Quadro 12 - Função remove
Fonte: Doctrine Documentation

2.6.13 Doctrine Query Language

DQL significa *Doctrine Query Language* e é um derivado de *Object Query Language* que é muito semelhante ao *Hibernate Query Language* (HQL) ou o Java *Persistence Query Language* (JPQL). É case *in-sensitive*, exceto para *namespace*, classes e nome de campo, que são *case-sensitive*(DOCTRINE REFERENCE, 2011).

DQL como uma linguagem de consulta tem *contructs SELECT*, *UPDATE* e *DELETE* que mapeiam para os seus tipos de instrução SQL correspondentes.

INSERT não são permitidos no DQL, porque as entidades e as suas relações têm de ser introduzidos no contexto de persistência através do *EntityManager#persist* () para garantir a coerência do seu Modelo de objeto.

Declarações *SELECT* no são uma forma muito poderosa de recuperação de partes do seu Modelo de domínio que não são acessíveis através de associações. Além disso, permitem recuperar entidades e suas associações em uma única consulta o que pode fazer uma diferença enorme no desempenho, em contraste ao uso de várias consultas.

Declarações *UPDATE* e *DELETE* oferecem uma forma de executar alterações em massa sobre as entidades do seu Modelo de domínio. Isso é muitas vezes necessário quando

você não pode carregar todas as entidades afetadas de uma atualização em massa para a memória.

O exemplo do Quadro 13, seleciona todos os usuários com idade > 20:

```php
<?php
$query = $em->createQuery('SELECT u FROM MyProject\Model\User u WHERE u.age > 20');
$users = $query->getResult();
```

Quadro 13 - Exemplo de select
Fonte: Doctrine Documentation

2.6.14 QueryBuilder

A *QueryBuilder* fornece uma API que é projetada para construir condicionalmente uma consulta DQL em várias etapas.

Fornece um conjunto de classes e métodos que é capaz de construir consultas de forma programada, e também oferece uma API fluente. Isso significa que você pode mudar entre uma metodologia para outra como quiser

Da mesma maneira que você constrói uma consulta normal, você constrói um objeto *QueryBuilder*, basta fornecer o nome do método correto. No Quadro 14 está um exemplo de como criar um objeto *QueryBuilder*:

```php
<?php
// $em instanceof EntityManager
// example1: creating a QueryBuilder instance
$qb = $em->createQueryBuilder();
```

Quadro 14 - Objeto QueryBuilder
Fonte: Doctrine Documentation

Depois de ter criado uma instância de *QueryBuilder*, ele oferece um conjunto de funções úteis que você pode usar. Um bom exemplo é verificar que tipo de objeto é o *QueryBuilder*.

Existem actualmente três possíveis valores de retorno para *getType* ():

- *QueryBuilder::SELECT*, quando retorna o valor 0
- *QueryBuilder::DELETE*, quando retorna o valor 1
- *QueryBuilder::UPDATE*, quando retorna o valor 2

O método *add*() é responsável por construir cada pedaço de DQL. Leva três parâmetros: *$dqlPartName, $dqlPart e $append (default=false).*

O Quadro 15 mostra um exemplo da utilização do *QueryBuilder*:

```php
<?php
// criação de uma instância QueryBuilder
$qb = $em->createQueryBuilder();

// como definir: "SELECT u FROM User u WHERE u.id = ? ORDER BY u.name ASC" usando o QueryBuilder
$qb->add('select', 'u')
   ->add('from', 'User u')
   ->add('where', 'u.id = ?1')
   ->add('orderBy', 'u.name ASC');

// Execute Query
$result = $query->getResult();
```

Quadro 15 - Usando o QueryBuilder
Fonte: Doctrine Documentation

2.6.15 Mapeamento XML

O *driver* de mapeamento XML permite fornecer os metadados ORM em forma de documentos XML. O *driver* XML é feito por um *XML Schema* que descreve a estrutura de um documento de mapeamento.

O documento de mapeamento de uma classe XML é carregado por demanda a primeira vez que é solicitado e, posteriormente, armazenado em *cache*. Todos os documentos de mapeamento devem começar com a extensão "*dcm.xml*" para identificá-lo como um arquivo de mapeamento *Doctrine*. Isto é mais uma convenção não sendo obrigatório.

2.6.16 Mapeamento YAML

O *driver* de mapeamento YAML permite que você forneça os metadados ORM em forma de documentos YAML.

O documento de mapeamento YAML de uma classe é carregado sob demanda à primeira vez que é solicitada e, posteriormente, armazenado em *cache* de metadados. Para funcionar, requer certas convenções:

- Cada entidade ou superclasse mapeada deve receber seu próprio documento YAML.
- O nome do documento de mapeamento deve ser constituído do mesmo nome da classe, onde os separadores de *namespace* são substituídos por pontos.

- Por convenção todos os documentos devem começar com a extensão "*dcm.yml*" para identificá-lo como um arquivo de mapeamento *Doctrine*.

2.6.17 Mapeamento PHP

Doctrine também permite fornecer os metadados ORM na forma de código PHP normal, utilizando a API *ClassMetadata*. Pode ser escrito diretamente no código PHP ou em arquivos dentro de uma função estática denominada *loadMetadata* (*$classe*) na própria classe de entidade.

2.6.18 Demarcação de Transação

Demarcação de transação é a tarefa de definir os limites de transação. A melhor demarcação de transação é muito importante porque se não feita corretamente, pode afetar negativamente o desempenho do seu aplicativo. Muitos bancos de dados e camadas de abstração de dados como o PDO, por padrão operam em modo *auto-commit*, o que significa que cada instrução SQL é envolvida em uma transação de pequeno porte. Sem qualquer demarcação de transação resulta em um desempenho ruim.

Para a maioria dos casos, *Doctrine* já cuida da demarcação de transação Todas as operações de *INSERT*, *UPDATE* e *DELETE* são colocadas em fila até que o *EntityManager#flush*() seja invocado e todas essas mudanças sejam feitas em uma única transação.

2.6.19 Sistema de Eventos

Doctrine possui um sistema de eventos leve que faz parte do pacote *Common*. O sistema de eventos é controlado pelo *EventManager*. É o ponto central de *listener* do sistema de eventos do *Doctrine*.

O sistema de eventos do *Doctrine* também tem um conceito simples de assinaturas de evento. Pode-se definir uma simples classe que implementa uma a interface que implementa um método que retorna uma série de eventos que devem ser subscritos.

2.6.20 Processamento em Lote

O controle de alterações é o processo que determina o que mudou em entidades gerenciadas desde a última vez, que foram sincronizadas com o banco de dados.

Doctrine oferece diferentes políticas de mudança de controle, cada um com suas vantagens e desvantagens. A política de controle de alterações pode ser definida com base em cada classe (ou mais precisamente, por hierarquia).

2.6.21 Objetos Parciais

Um objeto parcial é um objeto, cujo estado não é totalmente iniciado após ser reconstituído a partir do banco de dados e que está desconectado do resto de seus dados.

Doctrine, por padrão, não permite objetos parciais. Isso significa que, qualquer consulta que seleciona apenas os dados parciais de um objeto, irá levantar uma exceção dizendo que os objetos parciais são um problema. Se quiser forçar uma consulta para criar objetos parciais, possivelmente para ajustar o desempenho, pode ser usada a palavra-chave "*partial*" do seguinte modo visto no Quadro 16:

```php
<?php
$q = $em->createQuery("select partial u.{id,name} from MyApp\Domain\User u");
```

Quadro 16 - Uso da palavra-chave partial.
Fonte: Doctrine Documentation

Metadata Drivers

Doctrine fornece algumas maneiras diferentes para você especificar seus metadados:

- *XML files (XmlDriver);*
- *Class DocBlock Annotations (AnnotationDriver);*
- *YAML files (YamlDriver);*

- PHP Código em arquivos ou funções estáticas *(PhpDriver);*

Algo importante a se notar sobre os drivers acima é que todos resultam pra um mesmo fim. As informações de mapeamento são preenchidas com a instancia de *Doctrine\ORM\Mapping\ClassMetadata*. Assim, no fim, o *Doutrine* sempre apenas tem que trabalhar com a API da classe *ClassMetadata* para obter informações de mapeamento para uma entidade.

2.7 CODEIGNITER

O *CodeIgniter* é um kit de ferramentas para aplicações PHP. Seu objetivo é possibilitar que o desenvolvimento de projetos mais rapidamente.

Permite que se mantenha o foco no projeto minimizando a quantidade de código necessário para uma determinada tarefa. O *CodeIgniter* é disponibilizado sob uma licença de código aberto no estilo da *Apache/BSD.* Sendo assim, pode ser utilizado para qualquer fim.

Inicialmente escrito para ser compatível com o PHP 4 para atender a maioria dos servidores da época, não utilizando assim as vantagens do PHP 5 e ter um melhor gerenciamento de objetos. Atualmente após reestruturação do core, o *CodeIgniter* é totalmente compatível com o PHP 5 e não da mais suporte ao PHP 4.

CodeIgniter foi criado com os seguintes objetivos:

- **Instanciamento Dinâmico** - Os componentes são carregados e rotinas executadas apenas quando requisitadas. Nenhuma suposição é feita pelo sistema além dos mínimos recursos do núcleo, por isso o sistema é tão leve. Os eventos, disparados pela requisição HTTP, e os *Controllers* e *Views* determinarão o que deverá ser invocado.
- **Acoplamento flexível** - O quanto menos um componente depender do outro, mais reutilizável e flexível um sistema se torna.
- **Singularidade do Componente** - No *CodeIgniter*,cada classe e suas fonações são altamente autônomas a fim de serem as mais úteis possíveis.

2.7.1 Recursos do CodeIgniter

Por ser simples de instalar, a primeira vista o *CodeIgniter* pode parecer não atender as necessidades de desenvolvimento de uma grande aplicação. Porem ao analisar seus recursos, o mesmo se torna uma potente ferramenta ágil e simples.

A seguir está a sua lista de recursos:

- Sistema baseado na abordagem *Model-View-Controller*.
- Extremamente leve
- Classes completas de banco de dados com suporte a várias plataformas.
- Suporte a *Active Record Database*
- Validação de Formulários e Dados
- Segurança e Filtragem XSS
- Gerenciamento de Sessão.
- Classe de envio de e-mail. Suporte a *Anexox*, formatação HTML / Texto Puro, múltiplos protocolos (sendmail, SMTP, e Mail).
- Biblioteca de Manipulação de Imagens (recorte, redimensionamento, rotação etc.). Suporte a GD, *ImageMagick* e *NetPBM*.
- Classe para *Upload* de Arquivos
- Classes FTP
- Localização
- Paginação
- Encriptação de Dados
- *Benchmarking*
- Cache completo da página
- Log de Erros
- Perfis de Aplicação
- *Scaffolding*
- Classe Calendário
- Classe *User Agent*
- Classe *Zip Encoding*
- Classe de Gerador de Template
- Classe *Trackback*

- Biblioteca XML-RPC
- Classe de Teste de Unidade
- URLs amigáveis aos motores de busca
- Roteamento URI Flexível
- Suporte a Ganchos, Extensões de Classe e *Plugins*.
- Grande biblioteca de funções "*helpers*"

2.7.2 Fluxograma da Aplicação

O *index.php* serve como um controlador primário, iniciando os recursos básicos necessários para rodar o *CodeIgniter*. O roteador examina a requisição HTTP para determinar o que deve ser feito com ela. Se já existe o arquivo em *cache*, ele é enviado diretamente ao *browser*, pulando as outras etapas de execução. Antes do *Controller* de aplicação ser carregado, a requisição HTTP e dados submetidos pelo usuário são filtrados por segurança. O *Controller* carrega o *Model*, as bibliotecas principais, *plugins*, *helpers* e qualquer outro recurso necessário para processar a requisição específica. A *View* é renderizada e então enviada ao *browser*. A Figura 2 mostra como esse processo é realizado.

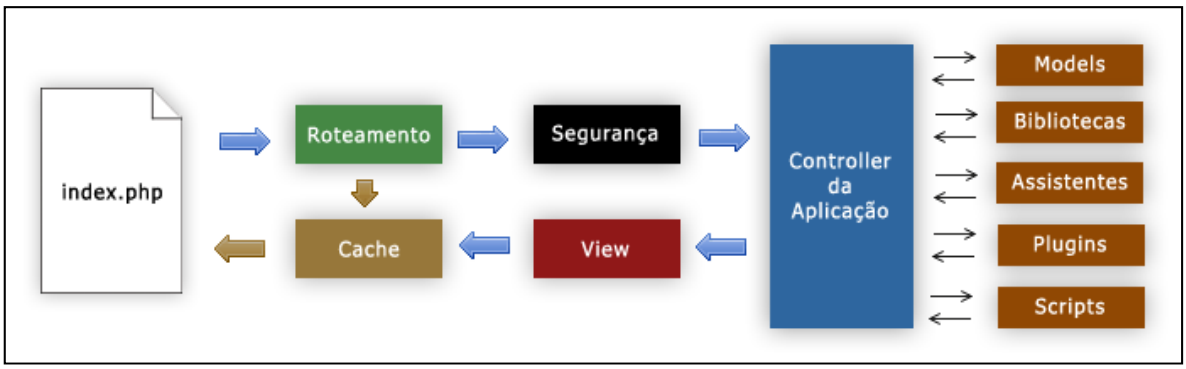

Figura 2 - Fluxograma da Aplicação
Fonte: CodeIgniter.

2.7.3 URLs Amigáveis

URLs no *CodeIgniter* são projetadas para serem amigáveis aos motores de busca e ao usuário. É usada uma abordagem baseada em segmentos. Por exemplo: *http://www.seudominio.com.br/classe/funcao/parametro*.

- **classe** - representa o *Controller* que será invocado;
- **função** - representa função ou método do *Controller* que será invocado;
- **parâmetro** - representa o parâmetro que será passado para a função;

Por default, o arquivo *index.php* será incluso em suas *URLs*, mas pode ser removido com a configuração de um arquivo *.htaccess*.

2.7.4 MVC

O *CodeIgniter* é baseado no padrão MVC que é uma abordagem de software que separa a lógica da aplicação de sua apresentação.

- ***Model*** - representa as estruturas de dados. Tipicamente, as classes *Model* irão conter funções que ajudam a buscar, inserir e substituir informações em banco de dados.
- ***View*** - é a informação que será apresentada ao usuário, que pode ser uma página *WEB* completa ou parte de uma.
- ***Controller*** - serve como um intermediário entre o *Model*, a *View*, e qualquer outro recurso necessário para processar a requisição HTTP e gerar a página *WEB*.

2.7.5 Helper Functions

São assistentes que ajudam em diversas tarefas. Cada *helper* é simplesmente um conjunto de funções de uma categoria particular. Há os Assistentes de *URL*, que ajudam a criar *links*, os Assistentes de Formulário que ajudam a criar os elementos de um *form*, os Assistentes de Texto que geram várias rotinas de formatação de texto, os Assistentes de *Cookie* que criam e lêem *cookies*, os Assistentes de Arquivo que ajudam a trabalhar com arquivos, entre outros.

2.7.6 Plugins

Os *plugins* trabalham de forma semelhante aos *helpers*. A diferença é que na maioria dos casos os *plugins* têm uma única função, enquanto os *helpers* são coleções de funções.

2.7.7 Libraries

Todas as bibliotecas disponíveis estão localizadas no diretório *system/libraries*. Na maioria dos casos, usar uma dessas classes envolve inicializá-la dentro de um *Controller*. Grande parte do poder do *CodeIgniter* se dá devido a suas classes e o que se pode fazer com elas. A seguir esta uma breve descrição de cada uma dessas classes:

- **Benchmarking Class**. A classe de *benchmarking* permite "marcar" pontos no código, fazer comparações sobre tempos de execução diferentes e até saber quanto de memória está sendo gasta para rodar a aplicação.
- **Calendar Class.** Com esta classe é possível criar calendários dinâmicos. Criar o *layout* dos calendários e inserir dados dinâmicos é tarefa simples com esta classe.
- **Cart Class**. A classe de carrinho de compras tem todo o básico que envolve processo de *e-commerce*: adicionar itens ao carrinho, atualizar, deletar, mostrar o preço, etc.
- **Config Class**. Serve para mostrar as opções de configuração do *CodeIgniter*, tanto as *default (/application/config/config.php)*, quanto as em arquivos personalizados.
- **Email Class**. Como sugere o próprio nome, esta é para envio de e-mails. Com extrema facilidade é possível adicionar endereços, CC, CCO, anexar arquivos e muito mais.
- **Encryption Class**. Se o problema é gerar strings encriptadas, então ele acabou. Com a classe de encriptação, as tarefas de encriptar, descriptar e correlacionados são bem mais simples.
- **File Uploading Class**. Com uma vasta gama de opções, fazer *upload* no *CodeIgniter* é a uma das coisas mais fáceis quando se usa a classe para *upload* de arquivos.

- **Form Validation Class**. Uma boa validação também deve ser feita *no back-end*; para tanto, o CI conta com muitas e diversificadas opções de validação para formulários.
- **FTP Class**. Classe para fazer as operações FTP tradicional, como transferências, mover arquivos, renomear, apagar, etc.
- **HTML Table Class**. Classe que permite gerar tabelas dinâmicas a partir de resultados de *arrays* e/ou consultas ao banco de dados.
- **Image Manipulation Class**. Uma das mais incríveis classes do CI permite fazer alterações em imagens como redimensionar, rotacional, criar miniaturas, inserir marcas d'água, dentre outros.
- **Input Class**. Tem dupla finalidade: pré-processar dados de inputs (questões de segurança) e funções para manipulação/verificação de dados via *input*.
- **JavaScript Class**. Classe que, usando funções *JavaScript*, provê algumas funcionalidades e efeitos básicos do *jQuery*, como *hide/show, fadeIn/fadeOut, animation* e outros.
- **Loader Class**. É a classe do *CodeIgniter* para carregar recursos diversos do *framework*, como *libraries, Views, helpers e Models.*
- **Language Class**. Classe que tem por objetivo prover suporte a internacionalização.
- **Output Class**. Classe que, juntamente com recursos de *cache*, tem a finalidade de enviar a *WEB page* requerida completa.
- **Pagination Class**. Classe muitos fácil de usar que permite fazer paginações em quaisquer conjuntos de resultados (muitas opções inclusas).
- **Security Class**. Como sugere o próprio nome, a *Security Class* provê funções para tratar da segurança da aplicação feita em CodeIgniter.
- **Session Class**. Classe para manipulação de sessões com o *CodeIgniter* (não são as sessões nativas do PHP).
- **Trackback Class**. Esta classe permite que se trabalhe com envio e recebimento de informações de *trackback*.
- **Template Parser Class**. Permite utilizar pseudo-variáveis nas *Views*, de modo a simplificar a separação HTML/PHP.
- **Typography Class**. Esta classe provê funções para formatação de textos.

- **Unit Testing Class**. Classe para se mexer com *Unit Testing* (conta com funções simples, mas possui várias opções).
- **URI Class**. Essencial para se trabalhar com CI, esta classe serve para se trabalhar com informações contidas em *URLs*.
- **User Agent Class**. Permite saber informações sobre o "agente" que acessa a aplicação *WEB* (*browser*, dispositivo *mobile*, robô de busca). Possui diversas opções interessantes.
- **XML-RPC Class**. Como consta no nome, esta classe serve para tarefas envolvendo XML-RPC.
- **Zip Encoding Class.** Permite tarefas com arquivos compactados *(.zip)*, tais como criação, adição/remoção de arquivos e até disponibilizar arquivos para download (forçado ou não).

2.7.8 Auto-load

O *CodeIgniter* vem com um recurso de "*Auto-load*" que permite que *libraries*, *helpers* e *plugins* sejam carregados automaticamente toda vez que o sistema rodar. Se você precisa de certos recursos globais em sua aplicação, deve considerar carregá-los automaticamente para sua conveniência.

Para carregar recursos automaticamente, abra o arquivo *application*/*config*/*autoload.php* e basta adicionar o item ao *array autoload*.

2.7.9 Tratamento de Erros

CodeIgniter permite criar relatórios de erros em uma aplicações utilizando as funções descritas abaixo:

- **show_error('messagem')** - Esta função irá mostrar a mensagem de erro disponível à ela utilizando o seguinte *template*: *application/errors/error_general.php*

- **show_404('pagina')** - Esta função irá mostrar a mensagem de erro 404 disponível à ela utilizando o seguinte *template*: *application/errors/error_404.php*
- **log_message('nivel', 'messagem')** - Esta função permite que se escreva mensagens nos arquivos de log. Basta fornecer um destes 3 “níveis” no primeiro parâmetro, indicando qual é o tipo de mensagem (*debug*, *erro*, *info*), e com a própria mensagem como segundo parâmetro.

2.7.10 Estrutura de Diretórios

Para trabalhar com o *CodeIgniter* é bom conhecer sua estrutura de diretórios, que é mostrada na Figura 3, ou seja, conhecendo como é a organização das pastas no *CodeIgniter* e sabendo suas funções dentro do “todo” é possível usá-lo com eficiência e consciência.

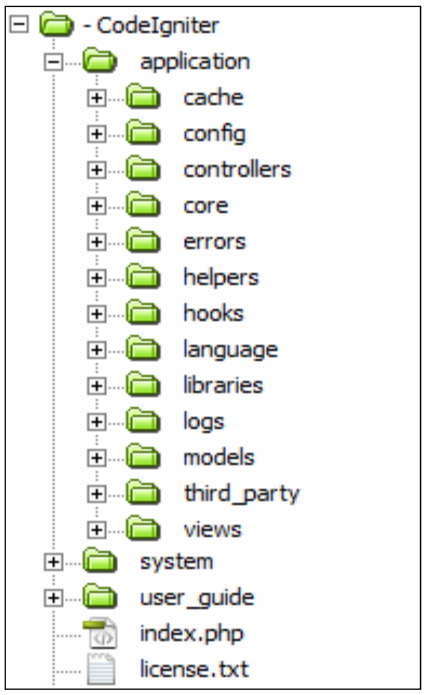

Figura 3 - Estrutura de Diretórios

Os três diretórios existentes na raiz da estrutura do *CodeIgniter* são:

- **application** - Pasta onde os arquivos do aplicativo desenvolvido ficam. Praticamente toda a codificação em *CodeIgniter* fica nesta pasta, que abriga seus.

- **system** - Local destinado aos arquivos do core. Geralmente, não devem ser alterados.
- **user_guide** - Contém o Guia do Usuário do *CodeIgniter*.
- Junto na raiz de diretórios do *CodeIgniter* existem dois arquivos:
- **index.php** - Primordial para o funcionamento do *CodeIgniter*, contém informações para se alterar o nível de *error reporting* que se vai trabalhar; opcionalmente, também é possível alterar os nomes padrão da pasta "*system*" e "*application*", dentre outras muitas configurações
- **licente.txt** - É o arquivo com a licença do *CodeIgniter* que deve constar em todo *software/aplicativo* feito em *CodeIgniter*.

2.7.11 Diretório Aplication

A pasta *application* é a mais importante para o desenvolvimento dos aplicativos e é a que mais vai exigir codificação. Já que a maioria do trabalho vai ser neste diretório:

- **cachê** - Pasta usada quando se está usando *caching* no *CodeIgniter*. É aqui que as páginas em cache ficam armazenadas e são servidas.
- **config** - Contém diversos arquivos relacionados a configurações do *CodeIgniter*. São arquivos de configuração de database, variáveis sobre URL, quais *libraries* e *helpers* serão carregados automaticamente e muitas outras coisas.
- **controllers** - Armazena os *controllers* criados.
- **core** - Pasta criada no *CodeIgniter* para conter algumas bibliotecas consideradas mais "core" que outras. Não é usada em todos os projetos.
- **errors** - Vem com os *templates* de páginas de erros do *CodeIgniter*. É conveniente que tudo isso seja alterado para que os erros do aplicativo fiquem personalizados e consonantes com a finalidade deste.
- **helpers** - Para armazenar todos os *helpers* criados.
- **hooks** - Para colocar os *hooks* criados.
- **language** - Para aplicativos multi-idioma, esta pasta é bem usada por armazenar as mensagens nas diferentes escritas.

- **libraries** - Aqui ficam as *libraries* personalizadas, com funcionalidades para o programa a ser criado. Perceba que há diferença entre as pastas *system/libraries e system/application/libraries*.
- **logs** - Quando se habilita o salvamento de *logs* em arquivo em */application/config/config.php,* é nesta pasta que eles serão salvos.
- **migrations** - Quando se usa acesso externo ao *CodeIgniter* via linha de comando (CLI), a pasta "*migrations*" é usada para algumas operações.
- **models** - Armazena os *models* criados.
- **third_party** - É possível distribuir aplicativos inteiros do *CodeIgniter* de forma fácil, e, para tal, é possível criar estrutura personalizadas nesta pasta, onde constam todos *helpers, liberaries, models* entre outras destas aplicações.
- **views** - Armazena os *views* da aplicação.

3 ESTUDO EXPERIMENTAL

Neste capitulo é apresentado um estudo experimental para demonstrar a utilização dos frameworks *Doctrine* e *CodeIgniter* de forma integrados.

Para este experimento foi desenvolvido um protótipo que consistiu em uma pagina *WEB* para avaliação de alunos onde, um usuário com nível de acesso privilegiado (administrador/professor) pudesse cadastrar questionários, que possuíam, além de questões de múltipla escolha, uma data de início e uma data de término. Quando iniciado o questionário, os usuários podem responder o mesmo e depois de confirmada a avaliação, é possível verificar o resultado.

Primeiramente, foi realizada uma configuração para que se pudesse trabalhar com o *Doctrine* dentro do *CodeIgniter*.

Os parâmetros para o desenvolvimento do protótipo foram:

- Ser organizado seguindo o padrão de projetos MVC;
- Utilizar *controllers* para encapsular as operações de persistência;
- Gerenciar as classes do modelo pelo framework *Doctrine* para suporte a transações e persistência;
- Gerenciar as funcionalidades de negócio pelo framework *CodeIgniter*;
- Utilizar PHP com o auxilio de HTML e CSS na programação da interface com o usuário.

Para análise e projeto do protótipo foi utilizada a linguagem UML (*Unified Modeling Language*), sendo que as etapas desenvolvidas foram:

- Descrição dos principais casos de uso e seus diagramas;
- Apresentação de um Modelo de Entidade Relacionamento;
- Desenvolvimento dos diagramas de seqüência;
- Especificação das entidades do negócio através de um diagrama de classes.

A apresentação das etapas do desenvolvimento se deu através de figuras que ilustram os formulários de controle da aplicação dentro de um navegador *WEB*.

Para o protótipo foram desenvolvidas cinco tabelas: *Usuarios; questionário; usuarios_has_questionarios; questoes; respostas; usuários_has_respostas*.

3.1 CONFIGURAÇÕES NECESSARIAS

A seguir estão os passos a serem seguidos para trabalhar com o *Doctrine* dentro do *CodeIgniter*:

O primeiro passo é criar um arquivo chamado *Doctrine.php* dentro do diretório *system/application/libraries.* Este será o arquivo de *bootstrap* que contém as configurações e parâmetros de funcionamento do *Doctrine*. Estas configurações são mostradas no Quadro 17.

```php
<?php
use Doctrine\Common\ClassLoader,
    Doctrine\ORM\Configuration,
    Doctrine\ORM\EntityManager,
    Doctrine\Common\Cache\ArrayCache,
    Doctrine\DBAL\Logging\EchoSQLLogger;

class Doctrine {

  public $em = null;

  public function __construct()
  {
    // carrega configuração do banco de dados para o CodeIgniter
    require_once APPPATH.'config/database.php';

    // Seta a classe de carregamento.
    require_once APPPATH.'libraries/Doctrine/Common/ClassLoader.php';

    $doctrineClassLoader = new ClassLoader('Doctrine',  APPPATH.'libraries');
    $doctrineClassLoader->register();
    $entitiesClassLoader = new ClassLoader('models', rtrim(APPPATH, "/" ));
    $entitiesClassLoader->register();
    $proxiesClassLoader = new ClassLoader('Proxies', APPPATH.'models/proxies');
    $proxiesClassLoader->register();
    // Configura os caches
    $config = new Configuration;
    $cache = new ArrayCache;
    $config->setMetadataCacheImpl($cache);
    $driverImpl = $config->newDefaultAnnotationDriver(array(APPPATH.'models/Entities'));
    $config->setMetadataDriverImpl($driverImpl);
    $config->setQueryCacheImpl($cache);

    $config->setQueryCacheImpl($cache);

    // Configuração de Proxy
    $config->setProxyDir(APPPATH.'/models/proxies');
    $config->setProxyNamespace('Proxies');

    // Configuração de Log
    $logger = new EchoSQLLogger;
    $config->setSQLLogger($logger);

    $config->setAutoGenerateProxyClasses( TRUE );

    // Informações da conexão com o banco de dados
    $connectionOptions = array(
        'driver' => 'pdo_mysql',
        'user' =>     $db['default']['username'],
        'password' => $db['default']['password'],
        'host' =>     $db['default']['hostname'],
        'dbname' =>   $db['default']['database']
    );

    // Cria um EntityManager
    $this->em = EntityManager::create($connectionOptions, $config);
  }
}
```

Quadro 17 - Configuração do arquivo Doctrine.php

Em seguida a pasta do *Doctrine* deve ser copiada para o diretório *application/libraries*. Para que o *Doctrine* seja carregado automaticamente com a aplicação, basta configurar o arquivo *systen/application/config/autoload.php*. Feito isso o *Doctrine* já pode ser utilizado nos *controllers* criados no *CodeIgniter*. Como mostrado no exemplo do Quadro 18.

```php
<?php
class UsuarioController extends Controller
{
    //Doctrine EntityManager
    public $em;

    function __construct()
    {
        parent::__construct();

        //Cria uma instancia do Doctrine Entity Manager
        $this->em = $this->doctrine->em;
    }

    function addUsuario(){
        $usuario = new models\Usuario;
        $usuario->setUsername('chicao');
        $usuario->setPassword('123');
        $usuario->setEmail('chicaoandrade@teste.com.br');

        $this->em->persist($usuario);
        $this->em->flush();
    }

}
```

Quadro 18 - Uso do Doctrine dentro de um Controller

3.2 RESULTADO E DISCUSSÃO

O protótipo desenvolvido possui dois principais atores: aluno e administrador. Os papeis de cada ator envolvido estão apresentados nos diagramas de casos de uso nas Figuras 4 e 5.

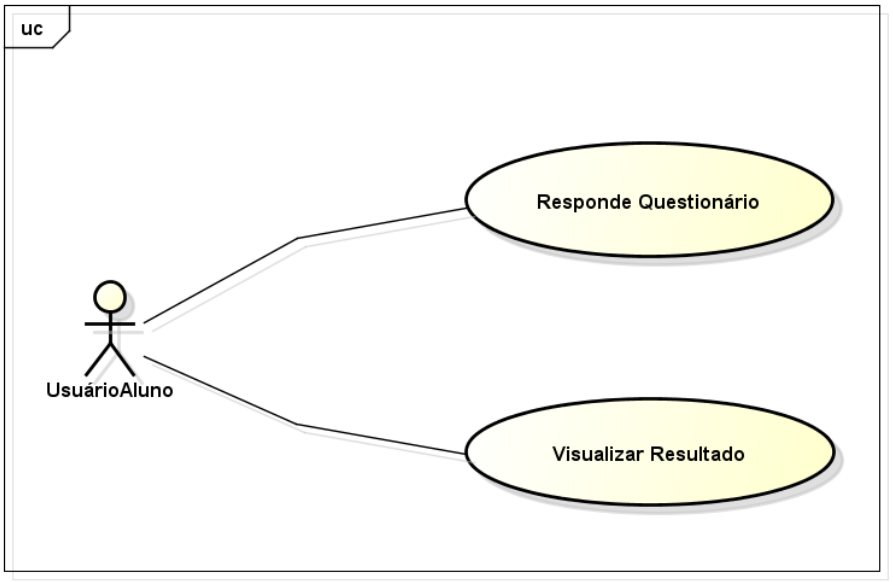

Figura 4 - Caso de uso para ator Usuário Aluno

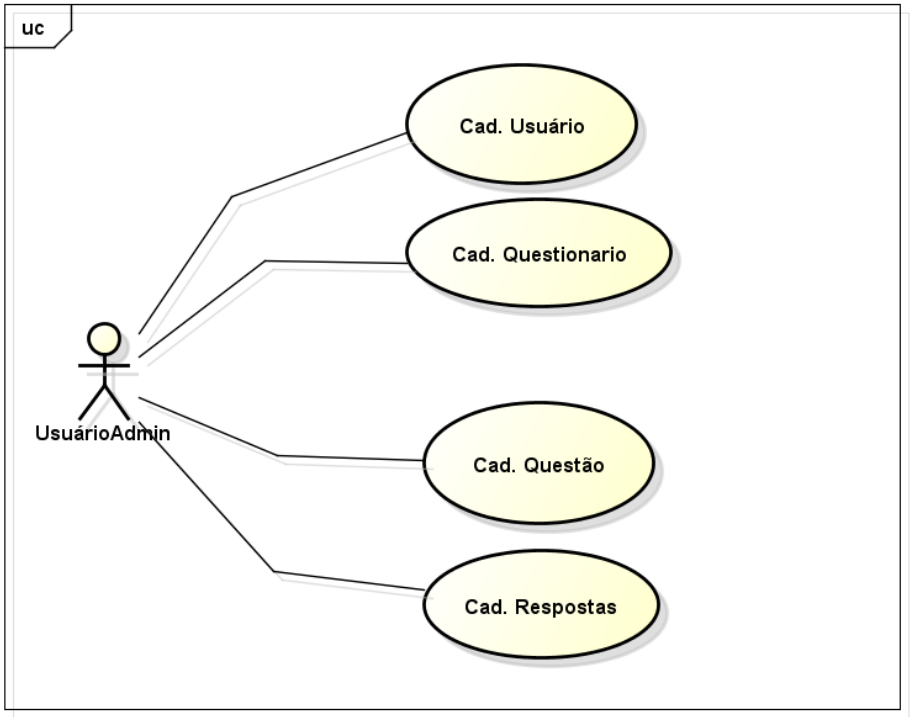

Figura 5 - Caso de uso para ator Usuário Admin

Os diagramas mostram que o usuário com acesso de aluno interage com o sistema, respondendo a questionários lançados e depois de realizada confirmação das respostas podendo visualizar o resultado da média alcançada e que o usuário do tipo administrador pode cadastrar novos usuários, cadastrar questionários, assim como questões e respostas para o mesmo.

As tabelas desenvolvidas para compor o banco de dados foram organizadas de forma que na tabela “usuários” ficassem armazenados dados de “usuárioadmin” (professores) e de “usuárioaluno”, distintos pelo nível de acesso. Na tabela “questionário” serão armazenados os questionários cadastrados com seu respectivo “idUsuario” que será o professor responsável. Na tabela “usuarios_has_questionarios” serão armazenados os registros de usuários referentes aos alunos e os questionários a eles destinados. Na tabela “questões” foram armazenadas as questões associadas a um questionário. Na tabela respostas serão armazenadas as respostas associadas a uma questão. E por fim, na tabela “usuários_has_respostas” ficarão armazenadas as respostas selecionadas pelo usuário. O Modelo de Entidade-Relacionamento do sistema proposto está descrito e detalhado na Figura 6.

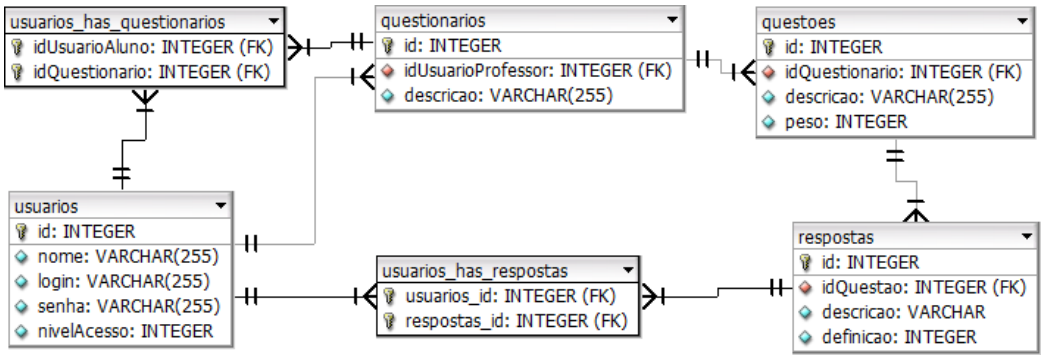

Figura 6 - MER do banco de dados

As entidades de negócio definem a relação entre as diferentes classes do protótipo. Estas entidades são anotadas, para persistência dos dados com *Doctrine* e, podem ser visualizadas no diagrama de classes apresentado na Figura 7.

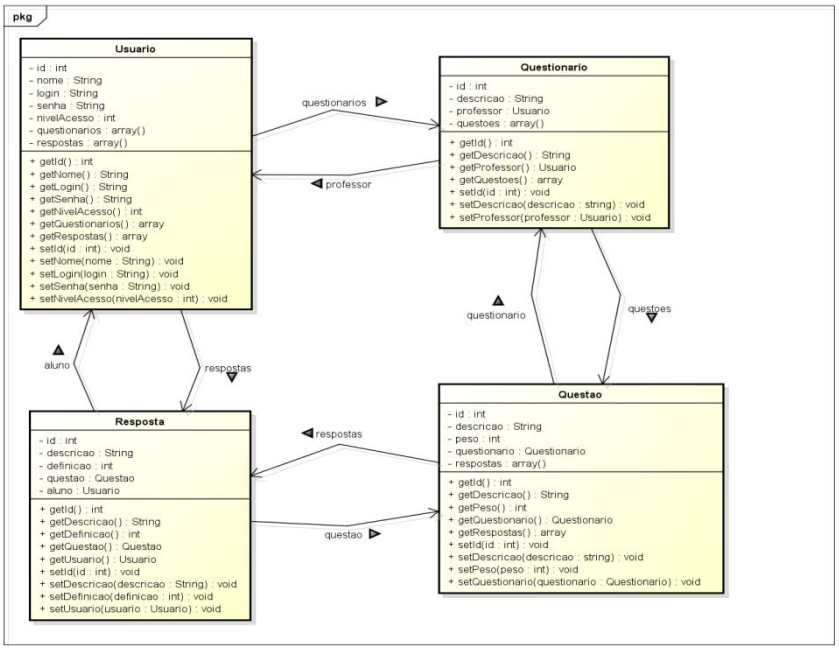

Figura 7 - Especificação das entidades de negocio

O mapeamento objeto relacional, é efetuado pelo *Doctrine* através de *Annotation*. A biblioteca *Doctrine/Common/Annotations* que permite que as informações de mapeamento sejam especificados como metadados incorporados nos arquivos de classe, sobre as propriedades e métodos. A biblioteca é independente e pode ser usado em suas próprias bibliotecas para a execução do bloco de anotações doc. A Figura 8 apresenta um código que efetua o mapeamento objeto relacional utilizando *Annotation*.

```php
<?php
namespace models;

/** @Entity */
/**
 * @Entity
 * @Table(name="usuarios")
 */
class Usuario
{
    /**
    * @Id
    * @Column(type="integer", nullable=false)
    * @GeneratedValue(strategy="AUTO")
    */
    private $id;

    /** @Column(type="string", length=255, unique=true, nullable=false) */
    private $nome;

    /** @Column(type="string", length=255, unique=true, nullable=false)) */
    private $login;

    /** @Column(type="string", length=255, unique=true, nullable=false) */
    private $senha;

    /** @Column(type="integer") */
    private $nivelAcesso;

    /**
     * @ManyToMany(targetEntity="Questionario")
     * @JoinTable(name="usuarios_has_questionarios",
     *      joinColumns={@JoinColumn(name="idUsuarioAluno", referencedColumnName="id")},
     *      inverseJoinColumns={@JoinColumn(name="idQuestionario", referencedColumnName="id")}
     * )
     */
    private $questionarios;
    //.....................
```

Quadro 19 - Classe Usuario

Iniciado o sistema o administrador acessa a sessão referente a usuários do sistema, que inicia com a listagem dos mesmos e das opções de inclusão de novo usuário ou edição de um já existente. Na figura 09 pode-se observar a seqüência inicial do sistema.

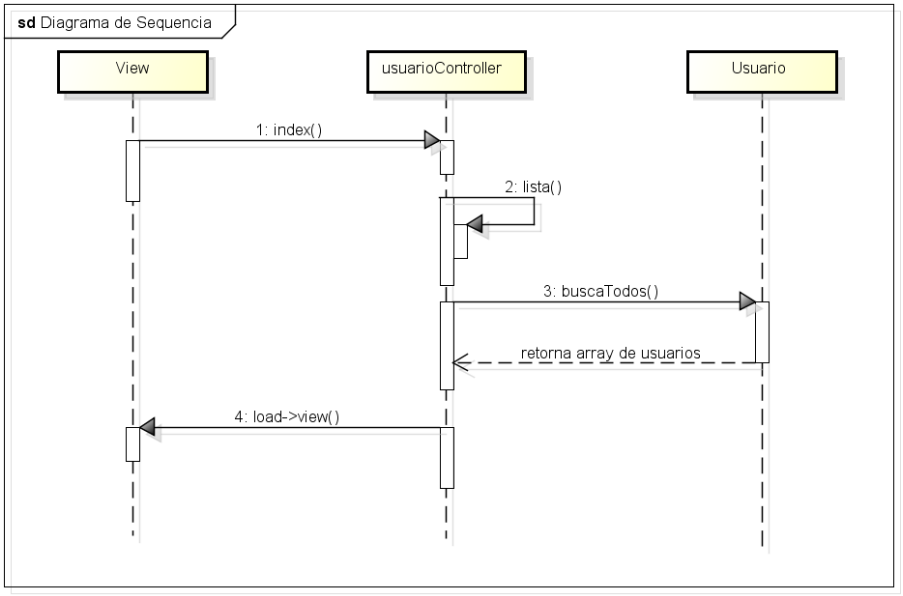

Figura 8 - Diagrama de seqüência lista de usuários

Observa-se que todas as entidades ficam submetidas a um *controller*. O *índex()* é chamado por *default* pelo *controller* ao iniciar a sessão referente ao usuário, que por sua vez chama outra função do *controller* chamada *lista()*. A *lista()* executa a chamada de uma função do modelo usuário fazendo uma busca de todos os usuário já cadastrados. O método *buscaTodos()* é a função do modelo responsável por buscar os registros de usuário alocando os mesmos em um *array()* e retornando o mesmo para o *controller* que chama a função *load->view(arquivo,data)* do *CodeIgniter* para carregar um arquivo de visualização. O *load->view('usuario\list',$data)* é chamado pelo *controller* e a pagina de listagem dos usuários é mostrada.

A listagem de usuários pode ser observada através de um *screenshot* na Figura 10, que também inclui *links* para a edição e inclusão de um novo usuário.

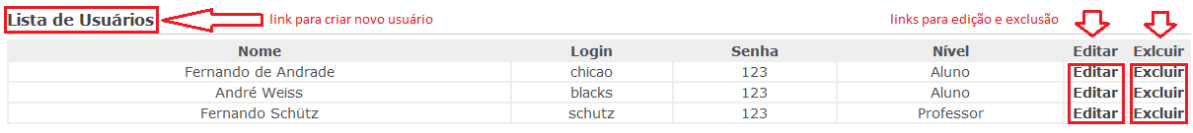

Figura 9 - Lista de usuários

Solicitada a inclusão de novo usuário será aberta a tela com um formulário para preenchimento dos dados, conforme Figura 11. Esse formulário é gerado com o auxilio do *helper* do *CodeIgniter* especifico para esta função.

Figura 10 - Formulário para inclusão de usuário.

Pode-se observar que o *helper form* do *CodeIgniter* facilita a construção de elementos HTML através de funções que auxiliam no trabalho com formulários. No Quadro 20, está descrito o código de exemplo do uso deste *helper*.

```
<div id="tabs">
    <? echo form_open(base_url().'usuarioController/save'); ?>
    <div id="tabs-1" class="tab-content">
        <fieldset class="ui-corner-all"><legend>Dados</legend>
        <input type="hidden" name="hdId" id="hdId" value="<?=$usuario->getId();?>" />
        <table cellpadding="0" cellspacing="0" border="0" class="form-table">
            <tr><th><label>Nome: </label></th>
                <td><?php echo form_input('txtNome', $usuario->getNome(),'id="txtNome", size="40"'); ?></td></tr>
            <tr><th><label>Login: </label></th>
                <td><?php echo form_input('txtLogin',$usuario->getLogin(),'id="txtDescricao", size="15"'); ?></td></tr>
            <tr><th><label>Senha: </label></th><td><?php echo form_input('txtSenha',$usuario->getSenha(),'id="txtSenha",
size="15"'); ?></td></tr>
            <tr><th><label>Nível Acesso: </label></th>
                <td><?php $nivel_selecionado = ($usuario->getNivelAcesso())?1:0;
                    $options = array("Aluno", "Professor");
                    echo form_dropdown('sltNivelAcesso', $options, $nivel_selecionado,'id="sltNivelAcesso"');?>
                </td></tr>
        </table>
        </fieldset>
        <fieldset class="ui-corner-all">
        <table cellpadding="0" cellspacing="0" border="0" class="form-table form-table-buttons">
            <tfoot>
            <tr>
                <td>
                    <input type="submit" name="btnEnviar" id="btnEnviar" value="Salvar" />
                    <input type="reset" name="btnLimpar" id="btnLimpar" value="Limpar" />
                    <input type="button" name="btnCancelar" id="btnCancelar" value="Cancelar" onClick=
"javascript:window.location='<?=base_url();?>usuarioController/lista';" />
                </td>
            </tr>
            </tfoot>
        </table>
        </fieldset>
    </div>
    <?
    echo form_close();
    ?>
</div>
```

Quadro 20 - Código de formulário

Para verificação na função *save()* chamada para salvar o novo registro, um campo *hidden* com o valor do id do registro é enviado junto com os dados do *POST*, caso o valor seja nulo indica que se trata de um novo registro. A função *save()* pode ser vista no Quadro 21.

```php
public function save(){

    $id = $this->input->post('hdId');

    if($id){
        $user = models\Usuario::busca($id);
    }else{
        $user = new models\Usuario();
    }

    $user->setNome($this->input->post('txtNome'));
    $user->setLogin($this->input->post('txtLogin'));
    $user->setSenha($this->input->post('txtSenha'));
    $user->setNivelAcesso($this->input->post('sltNivelAcesso'));

    $this->em->persist($user);

    $this->em->flush();

    $this->lista();

}
```

Quadro 21 - Função para inclusão de novo registro

Na função *save()* é feita a verificação sobre a variável *hdId* recebido via *POST*. No caso de novo usuário, é instanciado um objeto Usuario e a ele agregados os valores preenchidos no formulário, caso seja uma operação de edição de registro já existente, a variável *hdId* terá um valor indicando que a função *save()* deve fazer uma busca por código no banco de dados e retornar um objeto Usuario. O motivo de se fazer isso é permitir que com uma única função o sistema execute duas operações já que o *CodeIgniter* utiliza a função *persist($obj)* tanto para *insert* ou *update*, e também fica necessário apenas a criação de um arquivo, utilizando assim um único formulário. Feita a execução da função *persist($obj)*, novamente a função *listar()* é chamada e a página então é recarregada com os novos dados registrados em banco.

4 CONSIDERAÇÕES FINAIS

Com base no estudo realizado e nos resultados obtidos a partir da aplicação desenvolvida neste trabalho, pode-se chegar a algumas conclusões e sugestões para pesquisas futuras. Ambos apresentados neste capítulo.

4.1 CONCLUSÃO

Através desse estudo é possível concluir que, o uso de *frameworks* para o desenvolvimento de aplicações PHP é essencial para a obtenção de um bom resultado, visto que os *frameworks* estão bastante maduros e permitem a construção de aplicações cada vez mais flexíveis, além de tornar todo o processo de desenvolvimento mais rápido.

Para uma aplicação *WEB*, a integração entre os *frameworks Doctrine* e CodeIgniter se mostrou ser uma integração muito produtiva, tendo em vista que:

- O *CodeIgniter* é um *frameworks* com muitos recursos, mas que permite sua utilização modular. Dessa forma, utiliza-se somente o que for necessário para a aplicação, deixando-a mais.
- Utilizando o gerenciamento de transações do *Doctrine*, operações repetitivas de acesso a banco e sujeitas a erro de programação são evitadas.
- O *CodeIgniter*, se mostrou muito eficiente atuando na camada de visão. Existe uma variedade de *helpers* que implementam os componentes HTML de forma muito prática.

A aplicação desenvolvida no estudo de caso mostrou que os *frameworks* trabalham bem em sincronia, provendo consistência e robustez.

4.2 TRABALHOS FUTUROS/CONTINUAÇÃO DO TRABALHO

Este trabalho sugere uma pesquisa mais aprofundada no desenvolvimento de aplicações *WEB* utilizando os *frameworks Doctrine* e *CodeIgniter*, pois o número de recursos de desenvolvimento oferecido e grande, assim como a utilização de outros recursos que

podem auxiliar nesse desenvolvimento, por exemplo, o uso de *JavaScript* com *Jquery* e *Prototype* que tornam as aplicações mais flexíveis.

REFERÊNCIAS BIBLIOGRÁFICAS

ALLEN, Rob; LO Nick; BROWN Steven. **Zend Framework em Ação**. 1. Ed. Alta Books, 2009.

CodeIgniter Open Source. **User Guide.** 2011. Disponível em <http://codeigniter.com/user_guide/> Acessado em 19/05/2011.

Dall'Oglio, Pablo. **PHP Programando com Orientação a Objetos.** 2. Ed. Novatec, 2009.

Doctrine Project. **Reference Documentarion**. 2011. Disponível em < http://www.doctrine-project.org/projects/orm/2.0/docs/en> Acessado em 19/05/2011.

GABARDO, Ademir Cristiano. **CodeIgniter Framework PHP**. 1. Ed. Novatec, 2010.

GOLDBERG, Kevin Howard. **Desenvolvendo *WEB* sites dinâmicos com o CodeIgniter.** Disponível em <http://www.ibm.com/developerworks/br/opensource/library/os-codeigniter/>. Acesso em 01/09/2010

MANTOAN, Fernando Geraldo. **Integrando o Doctrine com o Zend Framework.** Disponível em <http://imasters.uol.com.br/artigo/16673/zend/integrando_o_doctrine_com_o_zend_framework/>. Acesso em 01/09/2010.

MYSQL. **Visão Geral do Sistema de Gerenciamento de Banco de Dados MySQL**. 2011. Disponível em <http://dev.mysql.com/doc/refman/4.1/pt/what-is.html> Acessado em 20/05/2011.

MINETTO, Elton. **O nascimento de um site com Codeigniter – Intro.** Disponível em <http://www.codeigniter.com.br/tutoriais/74/o-nascimento-um-site-com-codeigniter-intro>. Acesso em 01/09/2010.

PAUL, Michael. **Introdução a ORM em PHP com Doctrine**. 2009. Disponível em <http://www.michaelpaul.com.br/introducao-orm-php-doctrine.html> Acessado em 10/05/2011.

SOUZA, Fernando. **Uso de frameworks para desenvolvimento *WEB* e mitos que já deveriam ter desaparecido.** Disponível em <http://www.franciscosouza.com.br/2010/01/22/uso-de-frameworks-para-desenvolvimento-web-e-mitos-que-ja-deveriam-ter-desaparecido/>. Acesso em 02/09/2010.

Welling, Luke; Thomson, Laura. **PHP e MySQL Desenvolvimento *WEB*.** 3. Ed. Campus, 2005.