GOVERNO DO ESTADO DO RIO DE JANEIRO
SECRETARIA DE ESTADO DE CIÊNCIA E TECNOLOGIA
FUNDAÇÃO DE APOIO À ESCOLA TÉCNICA
CENTRO DE EDUCAÇÃO PROFISSIONAL EM TECNOLOGIA DA INFORMAÇÃO
FACULDADE DE EDUCAÇÃO TECNOLÓGICA DO ESTADO DO RIO DE JANEIRO
FAETERJ/PETRÓPOLIS

Desenvolvimento do Framework Lothus{PHP}

Guilherme Peixoto da Costa Louro

Petrópolis - RJ

Julho, 2015

Guilherme Peixoto da Costa Louro

Desenvolvimento do Framework Lothus{PHP}

Trabalho de Conclusão de Curso apresentado à Coordenadoria do Curso de Tecnólogo em Tecnologia da Informação e da Comunicação da Faculdade de Educação Tecnológica do Estado do Rio de Janeiro Faeterj/Petrópolis, como requisito parcial para obtenção do título de Tecnólogo em Tecnologia da Informação e da Comunicação.

Orientador:
Maria do Carmo Facó

Petrópolis - RJ

Julho, 2015

Folha de Aprovação

Trabalho de Conclusão de Curso sob o título *"Desenvolvimento do Framework Lothus{PHP}"*, defendida por Guilherme Peixoto da Costa Louro e aprovada em 31 de Julho de 2015, em Petrópolis - RJ, pela banca examinadora constituída pelos professores:

Prof. Maria do Carmo Facó
Orientador

Prof. Banca Interna
Faculdade de Educação Tecnológica do Estado
do Rio de Janeiro Faeterj/Petrópolis

Prof. Banca Interna
Faculdade de Educação Tecnológica do Estado
do Rio de Janeiro Faeterj/Petrópolis

Declaração de Autor

Declaro, para fins de pesquisa acadêmica, didática e tecnico-científica, que o presente Trabalho de Conclusão de Curso pode ser parcial ou totalmente utilizado desde que se faça referência à fonte e aos autores.

Guilherme Peixoto da Costa Louro
Petrópolis, em 31 de Julho de 2015

Dedicatória

Dedico esse trabalho a membros de minha família e amigos, principalmente a minha noiva por estar ao meu lado a todo momento me apoiando, mesmo nos momentos mais difíceis, nessa caminhada de dois anos e meio de faculdade e mais dois anos entre a criação do projeto e algumas pausas por motivos pessoais.

Gostaria de agradecer também aos que foram importantes em minha vida, me apoiando e motivando desde a escolha da faculdade até seus momentos finais.

Não podendo deixar de dedicar o trabalho aos companheiros de classe que viveram comigo os momentos fáceis e os mais complicados de toda a trajetória do curso, sem esquecer os que, de algum lugar, me passaram energia e motivação para a conclusão deste trabalho.

Agradecimentos

Ao meu orientador, professores e companheiros de trabalhor que se envolveram no desenvolvimento deste trabalho e deste projeto. Em especial agradeço a minha família e a minha noiva pela motivação e compreensão em momento difíceis e de ausência de minha parte em relação à dedicação dada a este projeto.

Epígrafe

"Quando acho que cheguei ao ponto máximo,
descubro que é possível superá-lo."

(Ayrton Senna)

Resumo

O mercado atual proporciona ao desenvolvedor diversas ferramentas, capazes de possibilitar a criação de sistemas de forma ágil e automatizada. Contudo, ferramentas como essas, dependem de um estudo baseado em suas documentações, a fim de se produzir um sistema de forma correta, sem fugir das determinações exigidas pela ferramenta utilizada. Esse processo, na maioria dos casos, exigem um certo tempo para o entendimento de todas as funcionalidades e métodos possíveis de serem usados, fazendo com que alguns desenvenvolvedores acabem testando outras ferramentas sem se especializar em nenhuma delas.

A proposta do projeto, apresentado neste documento é de um framework capaz de suprir as necessidades mais importantes no desenvolvimento de um site ou sistema web. De forma simples e sem a complexidade de ferramentas que já existem no mercado, o *Framework Lothus{PHP}* busca atender a todos os níveis de desenvolvedores, principalmente por ter uma curta curva de aprendizagem.

O *Framework Lothus{PHP}* utiliza MVC *(Model, View, Controller)* como padrão de projetos e foi produzido de forma a atender desenvolvedores FrontEnd e BackEnd, trazendo uma estrutura de pastas que separa cada etapa do processo de desenvolvimento.

Será detalhado todo o processo de desenvolvimento do *Framework*, mostrando todas as diretrizes e funcionalidades de cada etapa de sua criação. Este detalhamento será apresentado com explicações sobre arquivos e trechos de códigos pertencentes ao *Framework*, demonstrando, de forma transparente, o funcionamento de cada etapa.

Ao detalhar todo o processo de criação do *Lothus {PHP}*, serão descritos todas as ferramentas e tecnologias usadas para auxiliar o desenvolvimento capaz de tornar ágil futuras aplicações produzidas através do *Framework*.

Lista de Figuras

Sumário

1 Introdução

Devido à grande necessidade de entregar projetos de grande porte e com prazos consideravelmente baixos, foi percebida a necessidade de se encontrarem soluções que facilitassem esse desenvolvimento.

A primeira atitude a ser tomada foi a criação de um arquivo que reunisse diversas funcionalidades, a fim de facilitar futuros projetos, onde processos que se repetiam diversas vezes eram colocados em funções que poderiam ser usadas em novos projetos.

Aplicações, em geral, precisam de um padrão mais significativo como forma estrutural de um projeto, deixando claro que a criação de um arquivo contendo todas as funções do projeto não era o melhor padrão a ser seguido. Este trabalho apresenta, como uma de suas justificativas, uma pesquisa profunda que reúne novos padrões para os processos e *Frameworks* web que poderiam ser mais úteis para um desenvolvimento ágil.

No final dessas pesquisas iniciais, alguns *Frameworks* foram testados e o CakePHP passou a ser usado como padrão. O CakePHP utiliza o padrão de projeto MVC *(Model, View, Controller)* que é um modelo de arquitetura de software que tem como objetivo básico separar a lógica de negócio da aplicação.

Os *Frameworks* são sempre muito robustos e com diversos tipos de funcionalidades, e com o CakePHP não é diferente. Com uma vasta documentação e uma quantidade considerável de arquivos em seu projeto mais simples, este passou a ser um problema ao invés de solução quando se busca um total domínio em uma aplicação.

Percebeu-se, então, a real necessidade de criar um Framework onde se tenha total controle de todas as funcionalidades, mantendo o padrão MVC, porém criando as próprias funcionalidades, mesmo que baseado em funcionalidades de outros *Frameworks*.

Identificar o problema foi o primeiro e principal passo para se iniciar o desenvolvimento do *Framework*, que se encontra sempre em evolução com novas implementações que resolvam determinados problemas.

2 *A importância de se usar Framework*

Neste capítulo será apresentado, com dados técnicos, a importância do uso de um *Framework* em projetos de desenvolvimento, detalhando algumas de suas vantagens e desvantagens no processo de codificação.

O *framework* é, como princípio básico, uma arquitetura "padrão"que tem como objetivo fornecer ferramentas comuns a todo tipo de projeto, utilizando os mais variados tipos de Design Pattern (Padrões de Projeto) a fim de proporcionar um ambiente de desenvolvimento extremamente produtivo.

Grande parte dos *Frameworks* trabalham com um padrão principal denominado MVC (Model View Controller) que tem como base trabalhar com Modelo Lógico (Model), onde acontece toda a interação com a base de dados do projeto, Visualização (View), que é a parte responsável pela exibição de dados, e o Controle (Controller), que é a regra de negócios do projeto, pode-se dizer que o Controller é responsável por fazendo toda a comunicação com o Model e tratar os dados para serem exibidos pela View. Resumindo, o padrão MVC separa claramente o Design do Conteúdo e de sua Lógica.

2.1 Vantagens em usar um Framework

- **Padronização em projetos**: A grande vantagem de um *framework* é sua padronização no desenvolvimento. Por utilizar um conjunto já definido de Classes e Métodos, a necessidade de trabalhar conforme a ferramenta possibilita ajudar a garantir um aproveitamento maior de código projetos futuros.
- **Velocidade no desenvolvimento**: O fato de se fazer uso de módulos genéricos faz com que o *framework* fique responsável por controlar o uso de funcionalidades repetitivas fazendo com que o desenvolvedor se concentre totalmente na regra de negócios de cada projeto.
- **Qualidade**: *Frameworks* em geral são testados e atualizados a todo momento, tornado-se

cada vez mais seguros e com melhores funcionalidades.

- **Re-uso de códigos**: A padronização de projetos torna capaz o re-uso de código sem dificuldades de adaptação.
- **Segurança**: Uma das vantagens mais importantes é a segurança que o *Framework* pode dar ao projeto.
- **Fácil manutenção**: A separação do *framework*, utilizando padrões de projetos, permite uma fácil manutenção em determinada ferramenta sem que afete outras.
- **Utilitários e Bibliotecas**: Classes e métodos embutidos no *framework* para solucionar o problema de repetição contínua de códigos.

2.2 Desvantagens em usar um Framework

Esses pontos não são necessariamente uma desvantagem, porém são os principais motivos que inibem o desenvolvedor de iniciar em um *Framework*.

- **Performance e peso**: A grande quantidade de arquivo, a chamada de métodos e a criação de objetos nem sempre necessários para determinados projetos tornam a aplicação pesada em alguns casos.
- **Curva de aprendizado**: Ao se trabalhar com códigos de terceiros, existe uma curva de aprendizado elevada e que fica dependente de uma boa documentação para conseguir atingir um bom ritmo de trabalho.
- **Conhecimento técnico**: É necessário que se tenha conhecimento em OOP (Programação Orientada a Objeto), boas práticas de programação e se entendam padrões de projetos para poder utilizar o *Framework* da melhor forma.

3 *Experimento com Framework existente*

3.1 Cake PHP

3.1.1 Descricao da ferramenta

O CakePHP é um projeto de código aberto mantido por uma comunidade bastante ativa de desenvolvedores PHP. Possui uma estrutura extensível para desenvolvimento, manutenção e implantação de aplicativos. Utiliza o padrão de projeto MVC*(Model-View-Controller)* e ORM*(Object-relational mappring)* com os paradigmas das convenções sobre configurações.

3.1.2 Objetivo

CakePHP tem como objetivo principal a simplificação do processo de desenvolvimento e construção de aplicações web, utilizando um núcleo onde organiza o banco de dados e alguns recursos que reduzem a codificação pelo desenvolvedor. Alguns desses recursos são a validação embutida, ACLs *(lista de controle de acesso)*, segurança, manipulação de sessão e cache de Views e sanitização de dados.

3.1.3 Características

- Ativo e com comunidade amigável
- Compatível com PHP5
- Geração de CRUD (*Create, Read, Update and Delete, ou Criar, Ler, Atualizar e Excluir*)
- Funciona em qualquer subdiretório web, com poucas configurações no apache
- Utiliza templates

4 *Métodos e materiais*

Com o intuito de apresentar todo o processo de criação e utilização das ferramentas e funcionalidades do *Framework Lothus{PHP}*, neste capítulo serão apresentados e descritos os elementos e métodos utilizados para seu desenvolvimento e funcionamento.

4.1 PHP

O PHP (*um acrônimo recursivo para PHP: Hypertext Preprocessor*) é uma linguagem de script open source de uso geral, muito utilizada, e especialmente adequada para o desenvolvimento web e que pode ser embutida dentro do HTML.i(Php.net,2015).

O PHP contém um HTML com códigos embutidos que permite unir linguagem de marcação a códigos extremamente dinâmicos, utilizando as tags '**<?php**' e '**?>**' que separam o *HTML* do *PHP*.

É uma linguagem server-side (*lado do servidor*) e tem seu código executado diretamente no servidor, retornando somente o HTML que será exibido para o cliente, omitindo o acesso ao código PHP da aplicação.

4.2 POO *(Programação Orientada a Objetos)*

Programação Orientada a Objetos é um padrão de desenvolvimento com um conjunto de ideias, conceitos e princípios utilizados para facilitar e organizar melhor o desenvolvimento de aplicações. Tem como principais características facilitar a manutenção de códigos, utilizar com frequencia o reaproveitamento de códigos, além de diminiur a complexidade no desenvolvimento de sistemas.

4.2.1 Principais conceitos de POO

- **Abstração**: Utilizada para a definição de entidades do mundo real.

Entidade	Características	Ações
Carro, Moto	tamanho, cor, peso, altura	acelerar, parar, ligar, desligar
Elevador	tamanho, peso máximo	subir, descer, escolher andar
Conta Banco	saldo, limite, número	depositar, sacar, ver extrato

Figura 4.1: Abstração do mundo real.

- **Classes**: Definição dada para a estrutura de um objeto, onde são definidas os atributos e métodos referentes a cada objeto.
- **Objetos**: É a instância de uma classe. Um objeto é a construção de software que encapsula estado e comportamento, nos permitindo modelar a aplicação em termos reais e abstrações.
- **Herança**: É a possibilidade de uma classe *(subclasse)* herdar métodos e atributos de outra classe *(superclasse)*

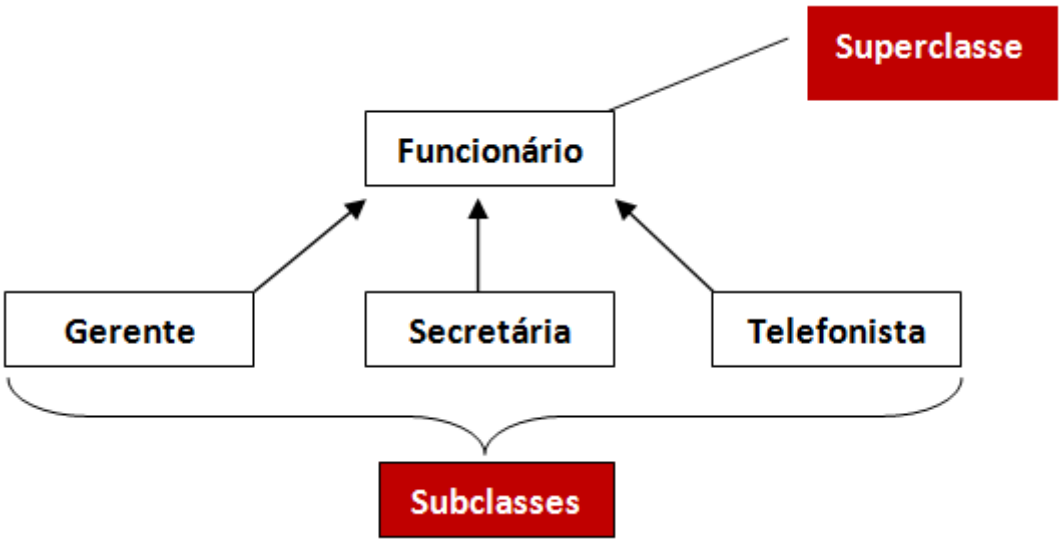

Figura 4.2: Hieraquia de classes

- **Polimorfismo**: Se trata da capacidade de um método ou comportamento da *superclasse* ser implementado de diversas maneiras nas *subclasses*.
- **Encapsulamento**: Forma de proteção dos atributos de uma classe, não permitindo que este seja acessado diretamente.

4.3 MVC *(Model, View e Controller)*

O MVC é um Design Pattern *(Padrão de projeto)* utilizado para separar as camadas de modelo, visão e controle no desenvolvimento de um sistema. A camada de modelo *(Model)* contém classes que implementam a regra de negócios da aplicação. A camada de visão *(View)*, por sua vez, é responsável pela exibição e apresentação dos dados para o usuário, e, por fim, a camada de controle *(Controller)*, onde são processadas todas as requisições realizadas pelos usuários.

A separação da aplicação em camadas, como é feita no padrão MVC, trás uma série de vantagens no processo de desenvolvimento: uma delas é a de permitir a reutilização do mesmo objeto de modelo em visualizações distintas, além de organizar seu projeto de forma onde tudo tenha seu lugar, e cada camada com sua responsabilidade, permitindo um trabalho muitos mais "centrado"e modularizado.

4.4 CRUD *(Create, Read, Update e Delete)*

CRUD é o acrônimo da expressão do idioma inglês, *Create Read Update and Delete* e é utilizado para designar as quarto operaçõs básicas de um banco de dados: Criar, Ler, Atualizar e Deletar.

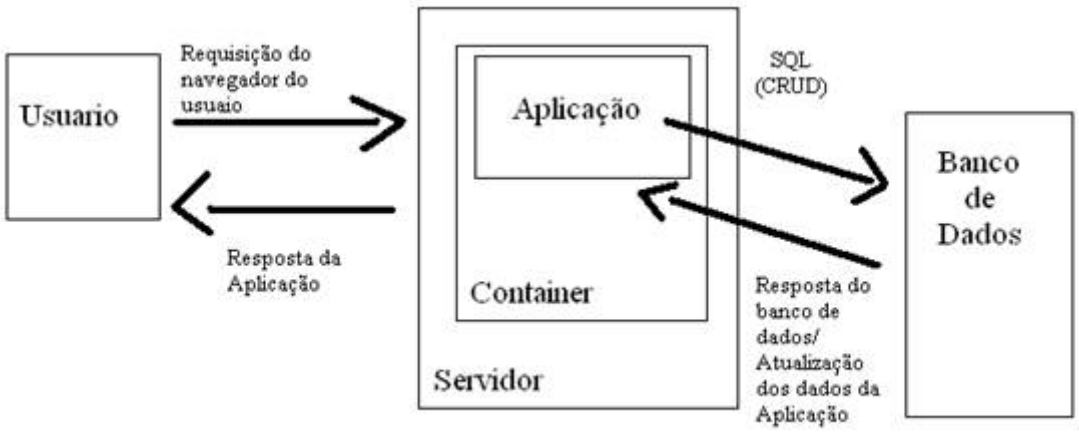

Figura 4.3: Execução de uma requisição CRUD

Tratando como uma forma mais técnica, o CRUD se transforma em um facilitador, criado através de diretivas de programação, para ações ligadas ao *INSERT*, *UPDATE*, *DELETE* e *SELECT* do banco de dados.

4.5 UML

UML *(Unified Modeling Language)*, é uma linguagem de modelagem que possibilita o desenvolvimento de diagramas de classes, de objetos, casos de uso, entre outros. Esses diagramas são úteis para o desenvolvimento e um grande facilitador para o entendimento de um projeto e sua estrutura.

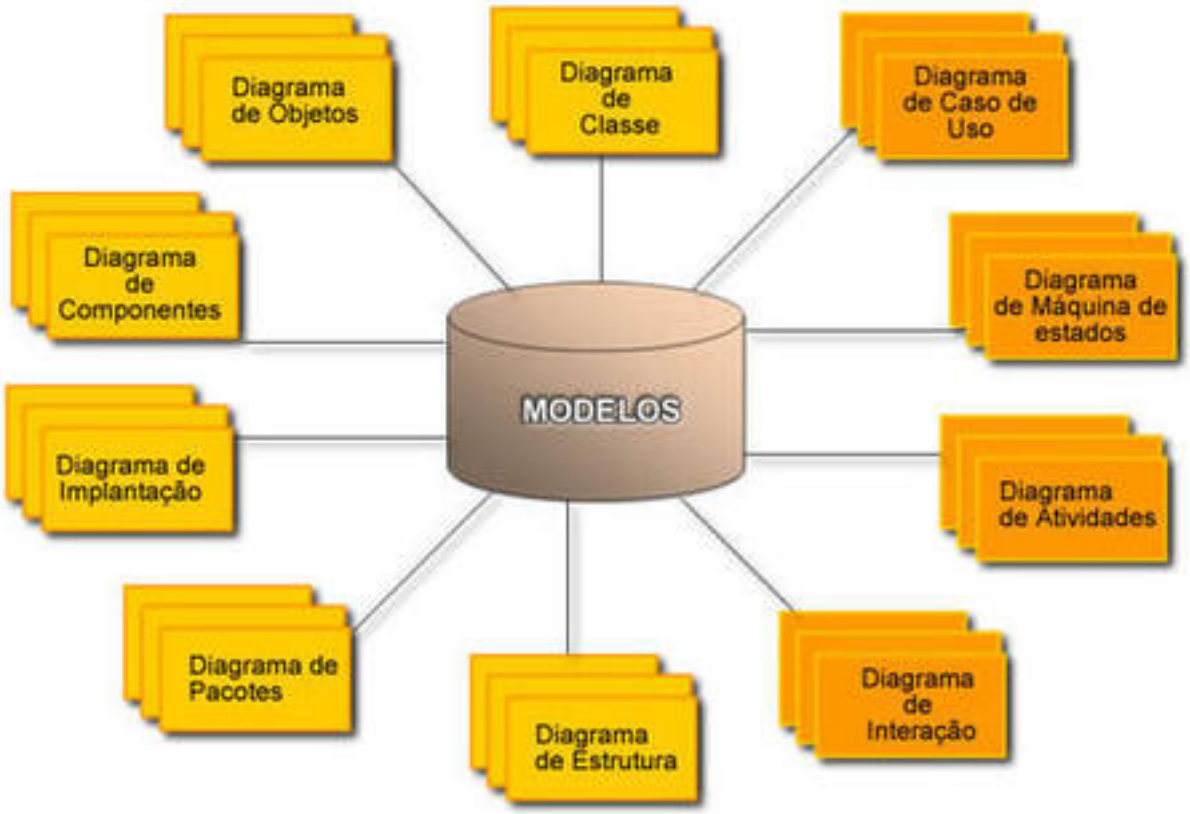

Figura 4.4: Exemplo de um diagrama UML.

4.6 Mysql

O MySQL é um gerenciador de banco de dados de código aberto, capaz de atender às necessidades dos mais variados tipos de usuários. Este produto tem uma gama diversificada de possibilidades de uso, algumas delas são soluções em desenvolvimento de sistemas, provedores, aplicações livres, entre outras.

4.7 PDO

PDO *(PHP Data Objects)* é um módulo de PHP montado sob o paradigma Orientado a Objetos e cujo objetivo é prover uma padronização da forma com que PHP se comunica com um banco de dados relacional. Este módulo surgiu a partir da versão 5 de PHP. PDO, portanto, é uma interface que define um conjunto de classes e a assinatura dos métodos de comunicação com uma base de dados. (LOCAWEB, 2015)

4.8 HTML

HTML *(Hyper Text Markup Language)*, como o próprio nome já diz, é uma linguagem de marcação de hipertexto utilizada no desenvolvimento de páginas web. Ela nos possibilita estruturar uma página através de marcações e tags específicas, permitindo que a mesma seja acessada pela internet.

4.9 Node.js

Node.js é uma plataforma construída sobre o motor de Javascript que tem como principal objetivo fornecer uma maneira fácil de se construir programas de rede escaláveis. Mesmo sendo um servidor de programas, não podemos confundi-lo com um servidor *ready-to-install*(prontos para instalar), que são servidores que estão prontos para instalar aplicativos instantaneamente. O *Node.js* segue o conceito de módulos que podem ser adicionados em seu núcleo. Há literalmente centenas de módulos para rodarem com o Node, e a comunidade é bastante ativa em produzir, publicar e atualizar dezenas de módulos por dia.

4.10 Automatizador Grunt

Grunt é uma ferramenta que roda via termina e serve para automatizar tarefas de uma aplicação, como: concatenação, minificação e validação de arquivos, otimização de imagem, testes unitários, deploy de arquivos por ftp ou rsync, entre outras. O *Grunt* é feito totalmente em *Javascript* e roda no *Node.js*, portanto para ser utilizado, depende da instalação do *Node.js* e do pacote *NPM* previamente instalados.

4.11 Sass

É um pre-processador de folhas de estilo feito em *Ruby* e responsável em auxiliar na produtividade de códigos *CSS*. Literalmente falando, *Sass* é uma extensão do *CSS* que adiciona potência e elegância à linguagem básica. Ele permite ao desenvolvedor o uso de variáveis, mixins, importações, ampla organização do código, entre outras funcionalidade totalmente compatíveis com *CSS*. *Sass* trabalha com dois tipos de *sintaxes* diferentes, *.sass* e *.scss*, e suas particularidas são: enquanto no arquivo *.scss* são utilizados chaves *"{}"* e ponto e vírgula *";"* para delimitar o inicio e fim de atributos e valores, no *.sass* essa delimitação é feita apenas por identação. Para ser utilizado em uma aplicação em produção utiliza-se o arquivo *CSS* gerado através da compilação do arquivo *.sass* ou *.scss*.

Sass

```
// variaveis
$bg_color: #333
$color: #fff
$space: 15

//placeholder
%hoverbt
  color: $color
  background-color: $bg_color

body
  // herança
  @extend %hoverbt
  font-size: sans-serif
  a
    color: $color
    &:hover
      color: darken($color, 5%)

// diretivas
@for $i from 1 through 3
  .space-#{$i*$space}
    margin-top: #{$i*$space}px
```

CSS

```
body {
  color: #fff;
  background-color: #333;
}

body {
  font-size: sans-serif;
}
body a {
  color: #fff;
}
body a:hover {
  color: #f2f2f2;
}

.space-15 {
  margin-top: 15px;
}

.space-30 {
  margin-top: 30px;
}

.space-45 {
  margin-top: 45px;
}
```

Figura 4.5: Compilação de um arquivo sass para css

4.12 Uglify

É um modulo que funciona em *NodeJs* responsável pela minificação e compressão de arquivos *Javascript*. Minificação consiste em reduzir o código, deixando-o apenas com o que é

necessários para seu funcionamento, sem afetar nenhuma funcionalidade.

4.13 Rsync

Rsync é uma ferramente que funciona apenas em sistemas *Unix*, responsável por transferência de arquivos, e capaz de sincronizar diretórios tanto locais quanto remotos. O *Rsync* pode transferir arquivo *Local -> Local*, *Local -> Remoto*, *Remoto -> Remoto*, *Remoto -> Local*. Ele trabalha sobre o protocolo SSH e *remote-update*, o que aumenta consideravelmente a velocidade e diminui a quantidade de dados transferidos, pois são trocados entre os servidores somente as diferenças entre dois grupos de arquivos reduzindo, também, o consumo de banda, além de ser muito mais seguro.

4.14 Controle de Versão: Git

É um sistema de controle de versão distribuído e open source que registra as mudanças feitas em um ou mais arquivos de forma que você possa recuperar versões específicas. Ele nos permite reverter arquivos ou até projetos inteiros para um estado anterior, comparar mudanças feitas com o tempo, ver qual desenvolvedor alterou determinado arquivo que pode estar causando problemas, entender em que ponto do projeto surgiu determinada falha no sistema, entre outras funcionalidades.

4.15 Github

Github é um repositório online que utiliza o *Git* como controle de versão e armazena diversos projetos, facilitando em processos de instalação e permitindo colaboração de outros desenvolvedores.

4.16 Bootstrap

É um framework de front-end que tem o objetivo de facilitar o desenvolvimento de interfaces para web. Contém uma coleção de vários elementos e funções personalizáveis para projetos da web, empacotados previamente em uma única ferramenta. Por se tratar de um software livre, todos os seus elementos são personalizáveis e utilizam uma combinação de HTML, CSS e Javascript.

4.17 Javascript

É uma linguagem de programação *client-side* utilizada para controlar *HTML* e *CSS* manipulando comportamentos e elementos de páginas web.

4.18 Jquery

É uma biblioteca que tem como objetivo simplificar tarefas complexas da programação em *Javascript*. Sua intenção é que fazer com que o desenvolvedor codifique menos, porém tenha o mesmo, ou um melhor, resultado sobre determinada ação.

Entre as suas características principais, a biblioteca jQuery contém:

- Manipulação do HTML/DOM;
- Manipulação de CSS;
- Métodos de eventos HTML;
- Efeitos e animações;
- (*AJAX*) Ferramenta Jquery para trocar de informações com servidor sem precisar atualizar a página web atual;
- Entre outras funcionalidades genéricas.

5 *Estrutura e funcionamento*

5.1 Processo de instalação

O *Framework Lothus{PHP}* permite ao desenvolvedor a possibilidade de escolha entre dois níves de aplicação. O primeiro nível permite a instalação do *Framework* da forma mais simples, instalando utilitário focados em um desenvolvimento direcionado ao backend do projeto, integrando facilidade à troca de informações com o banco de dados, desenvolvimento através do MVC, URLs amigáves e sistemas de templates.

Essa instalação é feita através do repositório remoto *Github*, que se encontra no seguinte endereço online:

https://github.com/guilouro/Lothus-PHP

O Github permite duas formas de download de um projeto: fazendo o downloand de um arquivo comprimido em .zip diretamente do site, ou utilizando um sistema de versionamento de arquivos para fazer o clone dele. Neste projeto iremos usar o *Git* como sistema de versionamento. Para fazer o clone utilizando o git, executamos a seguinte linha de comando no terminal Unix ou cmd Windows:

$ git clone https://github.com/guilouro/Lothus-PHP.git

Ao executar essa linha de comando, uma nova pasta será criada com o nome de Lothus-PHP. Dentro desta nova pasta estará todo o projeto para iniciar o desenvolvimento, utilizando o *Framework Lothus{PHP}*. O próximo capítulo será responsável pela apresentação das pastas existentes dentro do projeto.

5.2 Estrutura de pastas e arquivos

Neste capítulo será apresentada a estrutura de pastas do *Framework Lothus{PHP}* juntamente com o processo de criação e funcionamento de cada delas.

Ao clonar o projeto utilizando o git, como visto no capítulo anterior, será gerada uma estrutura de pastas dentro da pasta Lothus-PHP.

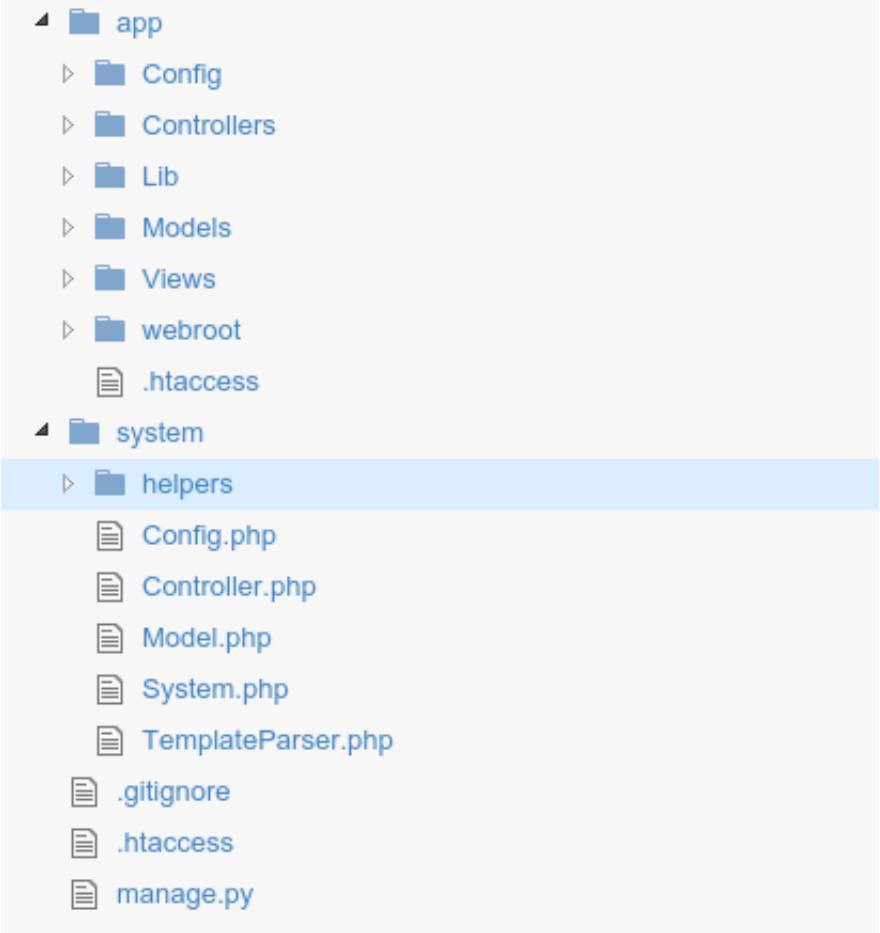

Figura 5.1: Estrutura do projeto

Neste primeiro momento, já ocorre uma pequena divisão do projeto no qual a pasta *app* é responsável por gerenciar a aplicação e a pasta *system* responsável pelo *core*, ou seja, gerenciamento interno do framework. Em sua raiz existem, além dessas duas pastas, três importantes arquivos para o projeto, que são:

- **.gitignore**: Um arquivo que faz parte da configuração do git e é responsável por guardar, linha por linha, todos os arquivos ou pastas ignorados pelo git no momento de fazer o versionamento do projeto. Isso evita o acúmulo de arquivos desnecessários, que são gerados automaticamente, no pacote de instalação do Framework.
- **.htaccess**: Arquivo que é lido antes do index.php e tem a responsabilidade de criar a rota inicial do projeto, fazendo o direcionamento para o arquivo correto na inicialização do sistema.

```
<IfModule mod_rewrite.c>
   RewriteEngine on
   RewriteRule    ^$ app/webroot/    [L]
   RewriteRule    (.*) app/webroot/$1 [L]
</IfModule>
```

Figura 5.2: Estrutura do htaccess na raiz do projeto

- **manage.py**: rata-se de um script de linha de comando capaz de gerar novos arquivos baseados na arquitetura de funcionamento do *Framework*. A imagem abaixo ilustra o uso básico da ferramenta.

```
================================================================================
Script que facilita a criação de {Controllers, Views e Models} para o Framework
================================================================================

 Uso básico:  manage.py [-name NOME]]


       -help           Exibe o painel de ajuda
       -name           Passa o argumento seguido do nome dos arquivos: manage.py [-name NOME]
       -table          Caso o nome da tabela no banco de dados seja diferente do passado no argumento -name
use esse argumento para passar o nome da tabela: manage.py [-name NOME] [-table NOME_TABELA]
       -c              Para criar apenas o controller manage.py [-name NOME] [-c]
       -m              Para criar apenas o model: manage.py [-name NOME] [-m]
       -v              Para criar apenas a view: manage.py [-name NOME] [-v]
       -v-name         Para escolher um nome específico para o arquivo phtml da view: manage.py [-name NOME]
 [-v-name NOME_VIEW] (não coloque o '.phtml')
```

Figura 5.3: Regras para uso básico do manage.py

5.3 Sistema

O Framework recebe um primeiro nível de divisão no processo de criação do mesmo, que é a divisão do Sistema da Aplicação. O sistema fica todo centralizado na pasta *system*, e é onde há o motor do *Framework*. Nele estão todas as classes responsáveis pelas regras de funcionamento do projeto, tanto nas requisições HTTP, passando por padronização de Controllers, Views, até chegar ao relacionamento com o banco de dados. Todas essas funcionalidades estão divididas entre classes e arquivos que serão detalhados ao longo deste capítulo.

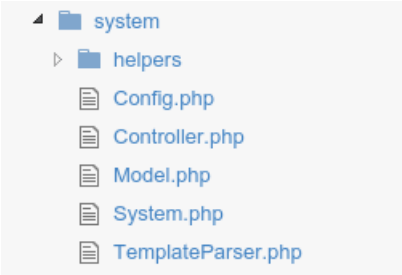

Figura 5.4: Estrutura interna da pasta system

5.3.1 Config

A Classe *Config* é responsável pela configuração inicial de qualquer projeto que faz uso do *Lothus{PHP}*. Ela tem a responsabilidade de definir qual *View* será iniciada ao acessar o link do sistema e também se responsabiliza em definir se será exibido ou não um *debug* para o desenvolvedor.

O arquivo *Config.php* possui a seguinte estrutura:

```
class Config {
    public  $_Index = "home";
    private $error = TRUE;

    protected function ERROR($pag){}
}
```

- **$_Index**: É uma variável **pública** que recebe, como string, o nome do *Controller* padrão a ser requisitado pelo sistema no caso de a URL não ter, explicitamente, este valor.
- **$error**: Trata-se de uma variável booleana **privada**, que funciona como uma chave para exibir um erro para o desenvolvedor ou direcionar o usuário para uma página 404, no momento em que for acessada alguma página inexistente. No caso de **$error = TRUE** será exibido uma mensagem de alerta ao desenvolvedor sobre alguma falha nos padrões do *Framework*. Caso **$error = FALSE** o usuário será redirecionado para uma página de *erro 404*
- **ERROR($pag)**: É um método protegido, que recebe como parâmetro o nome da página que não foi encontrada no sistema. Sua funcionalidade é, inicialmente, verificar se a variável **$error** é **TRUE** *(Verdadeiro)* ou **FALSE** *(False)*, para posteriormente, direcionar o usuário para a página de erro padrão do sistema ou exibir uma mensagem dizendo se o erro foi causado pela falta de um *Controller* ou de uma *Action* para o sistema.

A lógica de programação aplicada a este método é a seguinte:

```
protected function ERROR($pag) {
    if($this->error) {
        /* Erro em Controller ou Action */
    } else {
        /* Redirecionamento */
    }
}
```

5.3.2 System

O projeto é iniciado com a primeira chamada sendo referenciada à classe *System* que tem como herança os métodos e atributos, que não são privados, da classe *Config*. Sua principal funcionalidade é interpretar o padrão de URL criado para o Framework e fazer a separação para a camada correta de Controllers e actions com seus respectivos parâmetros, quando houver.

O *Lothus{PHP}* usa um padrão de url denominado **url amigável**, que facilita tanto a leitura dos mecanismos de buscas quanto a leitura do próprio usuário, além de padronizar todos os projetos desenvolvidos pelo *Framework*.

A URL segue o seguinte padrão:

http://urldosite.com.br/{Controller}/{Action}/{n-parametros}

- **Controller:** Classe controller referente à página acessada
- **Action:** Método existente nessa mesma classe
- **n-parametros:** Será passado como argumento para o método desta mesma classe controller.

Todos esses itens serão abordados corretamento no momento em que for descrito o funcionamento das classes de Aplicação do projeto.

Pode-se explicar que a classe system, ao ser invocada, recebe a URL via GET e através de alguns métodos essa URL é desmembrada, as variáveis da classe são definidas e por fim o projeto se inicia, como demostrado na estrutura da classe logo abaixo.

```
class System extends Config {

    public $_url;
    private $_explode;
    public  $_controller;
    public  $_action;
    public  $_params;

    public function init(){}
    private function setUrl(){}
    private function setExplode(){}
    private function setSlug($w){}
    private function setController(){}
    private function setAction(){}
    public function setParams(){}
    public function setGets(){}
    public function run(){}
}
```

A classe possui alguns atributos responsáveis por guardar determinadas informações a serem utilizadas em diversas etapas de sua leitura. Esses atributos têm suas responsabilidades descritas abaixo.

- **$_url**: Atributo público que receberá a url atual como string.
- **$_explode**: É um vetor privado que receberá, em cada uma de sua posição, uma parte da URL que se utiliza da *"/"* como regra de separação.
- **$_controller**: Atributo público que guardará o nome do Controller a ser usado.
- **$_action**: Atributo público que guardará o nome da Action a ser usada.
- **$_params**: Vetor público que guardará todos os parâmetros passados pela URL.

Esses valores são atribuídos através dos métodos que além de atribuir, fazem uso dessas mesmas variável da classe, como descrito nos elementos de cada método abaixo.

- **init()**: Primeiro método a ser chamado, explicitamente, pela classe e responsável por fazer a chamada de todos os métodos que fazem as atribuições a todas as variáveis dessa classe.

```
public function init() {
    $this->setUrl();
    $this->setExplode();
    $this->setController();
    $this->setAction();
    $this->setParams();
    $this->setGets();
}
```

- **setUrl()**: Responsável por atribuir a string da URL atual para a variável **$_url**. Caso a URL passada não tenha, explicitamente um controller e uma action, será atribuído o valor da variável **$_Index**, que é uma herança da classe Config, juntamente com a action padrão **index_action**.

```
private function setUrl() {
    $this->_url = (isset($_GET['url']) ? $_GET['url']  :
        $this->_Index . "/index_action" );
}
```

- **setExplode()**: Método que atribui ao array*(Vetor)* **$_explode** os valores passados para a variável **$_url** e delimitados pela barra*("/")*.

```
private function setExplode(){
    $this->_explode = explode("/", $this->_url);
}
```

- **setController()**: Define qual controller será usado para a requisição atual, através do primeiro índice do vetor **$_explode**

```
private function setController(){
    $this->_controller = $this -> setSlug($this -> _explode[0]);
}
```

- **setAction()**: Define qual action será usada para a requisição atual, através do segundo índice do vetor **$_explode**. Caso não exista, será atribuído o valor **"index_action"**.

```
private function setAction(){
    $this->_action = $this -> setSlug(
        !isset($this->_explode[1]) ||
        $this->_explode[1] == null ||
        $this->_explode[1] == 'index' ? 'index_action' :
        $this->_explode[1]);
}
```

- **setParams()**: Responsável por criar o vetor para todos os parâmetros passados pela URL atual. Parâmetros esses que são definidos por todos os valores passados além do controller e action na string da URL.

```
public function setParams(){
    unset($this->_explode[0], $this->_explode[1]);
    if( end( $this->_explode ) == null )
        array_pop($this->_explode);
    $this->_params = $this->_explode;
}
```

- **setGets()**: Verifica a aparição da requisição GET no HTTP da página atual e define o vetor **$_GET** do PHP com seus respectivos valores.

```
public function setGets(){
    $url = $_SERVER['REQUEST_URI'];
    $url = explode("?", $url);
    if(isset($url[1])) {
        $urlParams = explode("&", $url[1]);
        foreach ($urlParams as $g) {
            $get = explode("=", $g);
            $_GET[$get[0]] = $get[1];
        }
    }
}
```

- **setSlug($w)**: É responsável por verificar se o item **$w** possui duas ou mais palavras separadas por ífem. Caso exista, esse método retira os hífens e transforma todas essas palavras em apenas uma no modo *CamelCase*.

```
private function setSlug($w){
    $arr = explode("-", $w);
    for ($i=1; $i < count($arr); $i++) {
        $arr[$i] = ucfirst($arr[$i]);
    }
    $slug = implode("", $arr);
    return $slug;
}
```

- **run()**: É o método responsável por carregar o controller e a action com todos os parâmetros, caso eles existam. Se algum desses objetos não forem encontrado, o método **$ERROR**, herdado da classe *Config*, é chamado para exibir a mensagem adequada ao usuário.

5.3.3 Helpers

São classes que auxiliam o desenvolvedor no andamento do projeto, podendo ou não ser utilizadas em determinadas aplicações. O *Framework* vem com algumas classes em sua instalação. Outras podem ser adicionadas, se o desenvolvedor sentir necessidade. As classes se encontram dentro da pasta **helpers** e segue um padrão de nomenclatura que é o nome da classe acompanhada da palavra *Helper*. Para uma classe chamada **Email**, teria que haver a seguinte nomenclatura: **EmailHelper**.

O projeto acompanha algumas classes que serão descritas logo abaixo:

- **AuthHelper**: Auxilia na autenticação de usuários para projetos que possuem áreas restritas. Ela se responsabiliza em fazer o login do usuário no sistema de forma segura, comparando a senha digitada com a que possui no banco de dados através da criptografia **sha512** e iniciando uma sessão para o usuário. Possui um método de logout, que finaliza uma sessão com o usuário e o desconecta da área restrita do sistema. Além de possuir métodos que ajudam no controle de páginas protegidas, fazendo a verificação da permissão cada vez que o usuário tenta acessá-las.

- **EmailHelper**: Classe auxiliadora para envio de emails. Extende à classe *PHPMailer* e trabalha como uma facilidadora para esta, que é uma ferramenta avançada para envio de emails em *php*.

- **ImageHelper**: Esta é uma classe para otimização e salvamento de imagens nos projetos. Ela pode trabalhar tanto com imagens enviadas por um formulário, quanto imagens provenientes de algum link da internet. Sua estrutura é formada apenas por um construtor e mais 3 métodos públicos. Os métodos **ResizeByUrl** e **ResizeByUpload** salvam o arquivo de imagem na pasta especificada através do construtor da classe e retornam o nome gerado para o arquivo pelo método **Rename**. Esse nome é gerado para evitar a criação de imagens caracteres especiais em seu nome, evitando, inclusive, sobrescrever imagens com o mesmo nome. A classe obedece à seguinte estrutura descrita abaixo:

```
class ImageHelper {

    private $_ALTURA_PADRAO;
    private $_pasta;

    /*
     * Metodo construtor
     * @param $pasta = Nome da pasta para upload
     * @param $altura_padrao = Definir uma altura padrão para os
    arquivos
     */
    function __construct($pasta, $altura_padrao = null) { }

    /*
     * Metodo ResizeByUrl
     * Salva uma imagem a partir de um link
     * @param $url = Url da imagem a ser salva
     * @param $altura = Definir uma altura para a imagem atual,
   caso seja diferente da altura padrão
     */
    public function ResizeByUrl($url, $altura = null) { }

    /*
     * Metodo ResizeByUrl
     * Salva uma imagem a partir de upload
     * @param $imagem = ex: $_FILES['imagem']
```

```
     * @param $altura = Definir uma altura para a imagem atual,
   caso seja diferente da altura padrão
     */
    public function ResizeByUpload($imagem, $altura = null) { }

    /*
     * Metodo Rename
     * Remove acentos, espaços e caracteres especiais e cria um
   nome aleatório.
     * @param $string = string a ser limpa
     */
    public function Rename( $string ) { }
}
```

- **PaginationHelper**: Classe capaz de dividir um conteúdo vindo do banco de dados em diversas páginas através de uma paginação. A classe permite, ao ser instanciada, receber o nome da tabela no banco de dados onde será feita a consulta. Essa consulta possibilita ao desenvolvedor escolher quantos itens serão exibidos por páginas, e a ordem de exibiçao. Além dessa exibição, ela fornece alguns métodos auxiliares onde se pode ter o retorno da página ou url atual. A classe possui uma estrutura bem simples, como mostrado abaixo.

```
class PaginationHelper
{
    private $_model;
    private $_limite;
    private $_paginaAtual;
    private $_inicio;
    private $_total_registros;

    /**
     * Classe para paginação
     * @param String $tabela tabela no banco de dados para a
   consulta
     */
    function __construct( $tabela ) { }

    /**
```

```
     * Pega a quantidade total de páginas de acordo com a lista
     * @return int retorna a quantidade de páginas
     */
    public function totalPaginas() { }

    /**
     * Responsável por fazer a consulta limitada
     * para a paginação e definição de alguns atributos
     * @param  Array $arr parâmetros SQL. 'where' , 'orderby'
     * @return Array      retorna a consulta limitada
     */
    public function consulta(Array $arr = null) { }

    /**
     * Retorna a página atual captura
     * @return int
     */
    public function getPaginaAtual() { }

    /**
     * Gera url atual
     * @return string retorna a url até a parte da pagina
     */
    public function getUrl() { }

    /**
     * Cria a view para exibição da paginação
     * @param  string $classe define a classe da div que engloba
    a paginação
     * @return string
     */
    public function view($classe = "pagination pagination-
   centered") { }
}
```

Ao chamar o método **view()** no template do sistema, um componente de paginação é

criado de acordo com a quantidade de páginas existente na sessão atual. Esse método, por padrão, cria o componente com uma classe de *css* referente ao framework *Bootstrap*. Com isso o componente passa a ter uma formatação adequada para uma paginação.

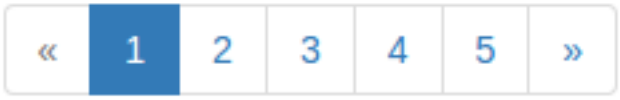

Figura 5.5: Componente de paginação criado por PaginationHelper

- **RedirectHelper**: Simples classe que fornece a possibilidade de trabalhar com redirecionamento dentro do projeto. Possui métodos para retorno de url, *Controller* atual e *Action* atual. Seus métodos principais são os de redirecionamento, tomando como base o *controller*, a *action* ou ambos como ponto de destino para este redirecionamento. A classe e seus principais métodos podem ser vistas no modelo abaixo:

```
class RedirectHelper {
    ...

    /**
     * Retorna o controller atual
     * @return (String)
     */
    public function getCurrentController() { }

    /**
     * Retorna a action atual
     * @return (String)
     */
    public function getCurrentAction() { }

    /**
     * Redireciona para o controller especificado
     * @param  (String) nome do controller
     * @return void
     */
    public function goToController( $controller ) { }
```

```
    /**
     * Redireciona para a action especificada
     * @param  (String) nome da string
     * @param  (boolean) para usar parâmetro global
     * @return void
     */
    public function goToAction( $action, $paramsGlobal = FALSE )
    { }

    /**
     * Redireciona para controller e action especificada
     * @param  (String) nome controller
     * @param  (String) nome action
     * @return void
     */
    public function goToControllerAction( $controller, $action )
    { }

    ...

}
```

5.3.4 Controller

Essa classe recebe a *System* como herança e possui, inicialmente, dois métodos e um atributo. Sua função é guardar o template que será usado e compilar a view para o sistema. A estrutura da classa está descrita abaixo.

```
class Controller extends System {

    public $_layout = 'default';

    protected function view($nome_pagina, $vars = null) { }

    public function init() { }
}
```

- **$_layout**: Atributo público para definir o template base da página a ser carregada.
- **view($nome_pagina, $vars ñull)**: Método responsável por atribuir o template da view ao template base, e recebe como primeiro parâmetro *($nome_pagina)* o nome do arquivo do template a ser utilizado. O Método possui um segundo parâmetro *($vars)* opcional, que é capaz de receber um vetor e distribuí-lo em variáveis para ser utilizada no template carregado.
- **init()**: É o primeiro método a ser chamado no momento da criação do Controller. Ele pode ser ou não implementado dentro de cada controller criado e é o primeiro método a ser executado sempre que carrega uma página.

5.3.5 Model

Classe responsável por fazer a comunicação entre aplicação e banco de dados. Utizando, como padrão, o módulo nativo do php chamado *PDO*. Esta classe segue a seguinte estrutura descrita abaixo.

```
class Model {
    protected $db;
    public    $_tabela;
    public    $_fk;

    public function __construct()  { }

    public function insert( Array $dados, $debug = FALSE ) { }

    public function read( $where = null , $limit = null ,
        $offset = null , $orderby = null, $debug = FALSE ) { }

    public function readLine( $where = null , $limit = null ,
        $offset = null , $orderby = null, $debug = FALSE ) { }

    public function update( Array $dados, $where, $debug = FALSE ) {
   }

    public function delete( $where ) { }

    public function consulta($sql, $debug = FALSE) { }
```

```
    public function consultaLinha ( $sql , $debug = FALSE ) { }

    public function consultaValor ( $sql , $debug = FALSE ) { }

    public function populateFK() { }

}
```

- **$db**: Atributo protegido que se torna uma instância da classe *PDO*, capaz de fazer o processo de envio e recebimento de dados entre aplicação e banco de dados. Este objeto é instanciado através do construtor da classe.
- **$_tabela**: Atributo público que guarda uma *string* como o nome da tabela, existente no banco de dados, que será manipulada pela classe.
- **$_fk**: Atributo que recebe um vetor, no caso de haver relações entre tabelas, ligadas com chave estrangeira.
- **insert(Array $dados, $debug=FALSE)**: Método público, responsável por fazer a inserção de novos dados na tabela que está especificada na classe. O metodo *insert* recebe, como parâmetro, um vetor, denominado **$dados**, com os dados a serem inseridos. Este vetor, por padrão precisa conter no mínimo uma chave e um valor; a chave será o nome da coluna e o valor será o conteúdo desta coluna a ser inserido no banco de dados. O retorno deste método é o *Id* da linha inserida caso a ação seja bem sucedida, caso contrário retornará um valor nulo. O método permite um segundo parâmetro, opcional, chamado **$debug**. Debug aceita valores *booleanos*; se receber **true** ele imprime na tela a string completa que faz a interação naquele momento com o banco de dados, no caso de **false**, ele não faz nada. Por padrão **$debud** é iniciado como falso.
- **read($whereñull, $limitñull, $offsetñull, $orderbyñull, $debug=FALSE)**: Método público, responsável pela consulta direcionada à tabela especificada na classe. Este método fornece ao desenvolvedor a possibilidade de passar alguns parâmetros, responsáveis pela filtragem dos dados retornados. A passagem de parâmetro é opcional e possui nomenclaturas intuitivas que auxiliam no entendimento das funcionalidades. Por padrão o método *read()*, sem passar parâmetro algum, retorna um vetor com dados relacionado a uma con-

sulta completa e sem filtros da tabela especificada. O retorno pode ser filtrado definindo os parâmentros do método.

  - **$where**: Condição a ser tratada pela clausula *WHERE* do mysql na consulta solicitada.
  - **$limit**: Responsável por determinar o ponto final, através da quantidade de linhas retornadas.
  - **$offset**: Responsável por determinar o ponto inicial, através da quantidade de linhas retornadas.
  - **$orderby**: Nome da coluna em que a consulta usa como base para ordenação do resultado.
  - **$debug**: Imprime na tela a string completa que faz a interação naquele momento com o banco de dados.

- **readLine($whereñull, $limitñull, $offsetñull , $orderbyñull, $debug=FALSE)**: Tem a mesma característica do método *read()*, porém o seu retorno é de um vetor com apenas um índice, baseado em uma consulta de uma linha do banco de dados.

- **update(Array $dados, $where, $debug=FALSE)**: Atualiza os dados em determinada linha de uma tabela, utilizado o *UPDATE* do mysql. O método possui a necessidade de passagem de dois parâmetros obrigatórios e dá a opção de um terceiro parâmetro não obrigatório, são eles:

  - **$dados**: Parâmetro obrigatório que possui um vetor com os dados a serem atualizados no banco de dados.
  - **$where**: Parâmetro obrigatório com a condição a ser utilizada pelo *WHERE* do mysql, com a responsabilidade de tratar em quais linhas da tabela serão feitas as substituições solicitadas.
  - **$debug**: Imprime na tela a string completa que faz a interação naquele momento com o banco de dados.

- **delete($where)**: Exclui uma linha do banco de dados referente à condição passada pelo parâmentro **$where**

- **consulta($sql, $debug=FALSE)**: Este método permite ao desenvolvedor passar uma string completa de sql para retornar um conjunto de linhas específicas a essa consulta. O método recebe as regras da consulta, faz a busca no banco de dados e retorna um vetor

com todas as linhas da tabela referente à consulta. Esse método não utiliza a variável $_tabela como referência ao banco de dados, e aceita dois parâmetros, que são os seguintes:

  - **$sql**: Parâmetro obrigatório que recebe a string completa de consulta ao banco de dados.
  - **$debug**: Imprime na tela a string completa que faz a interação naquele momento com o banco de dados.

- **consultaLinha($sql, $debug=FALSE)**: Este método permite ao desenvolvedor passar uma string completa de sql para retornar apenas uma linha referente a essa consulta. O método recebe as regras da consulta, faz a busca no banco de dados e retorna um vetor uma linha da consulta efetuada. Esse método não utiliza a variável $_tabela como referência ao banco de dados, e aceita dois parâmetros, que são os seguintes:
  - **$sql**: Parâmetro obrigatório que recebe a string completa de consulta ao banco de dados.
  - **$debug**: Imprime na tela a string completa que faz a interação naquele momento com o banco de dados.
- **consultaValor($sql, $debug=FALSE)**: Este método permite ao desenvolvedor passar uma string completa de sql para retornar apenas um valor referente a essa consulta. O método recebe as regras da consulta, faz a busca no banco de dados e retorna um único valor da consulta efetuada. Esse método não utiliza a variável $_tabela como referência ao banco de dados, e aceita dois parâmetros, que são os seguintes:
  - **$sql**: Parâmetro obrigatório que recebe a string completa de consulta ao banco de dados.
  - **$debug**: Imprime na tela a string completa que faz a interação naquele momento com o banco de dados.
- **populateFK()**: Caso a variável **$_fk** possua um vetor como valor, este método é capaz de percorrer cada valor, executar o método *read()* referente a cada tabela e popular o retorno para uma variável com o mesmo nome da tabela do banco de dados.

5.3.6 Template

O *Framework Lothus{PHP}* possui uma forma de se trabalhar com templates. Esse controle é feito através da classe *TemplateParser*. A classe herda os valores e métodos da classe *System*,

o que permite a manipulação e utilização de alguns dados. Os templates ficam localizados dentro da pasta *app*
Views
Layouts e por padrão o template principal é o *defaultphtml*. O *Framework* permite trabalhar com diversos templates, para isso existem algumas regras a serem seguidas. Primeiramente precisa ser criado um arquivo como o nome do template dentro da pasta *app*
Views
Layouts do projeto atual, por exemplo, **novo_template.phtml**. Em seguinda, é necessário inserir a tag *{conteudo}* no local onde serão exibidas as views dependentes deste template. E finalmente dizer quais actions do sistema farão uso desse template definindo a variábel de layout com o valor do novo template *$this->_layout = ńovo_template´*. Esta variável é definida dentro do método que será responsável pela action da página em questão.

A funcionamento dos templates é, verificar o template padrão a ser usado, através da variável *$_layout*, que pertence à classe *Controller*, que por padrão é *ḿdefault´*, em seguinda é feito a busca da tag *{conteudo}* dentro deste template e esta tag é, por fim, substituída pela view que será exibida.

5.4 Aplicação

O Framework recebe um segundo nível de divisão em seu processo de criação, que é a parte da App, onde será construída toda a aplicação. A aplicação fica toda centralizada na pasta *app*, que contem a divisão *MVC* da aplicação. Todas essas funcionalidades estão divididas entre classes e arquivos que serão detalhados ao longo deste capítulo.

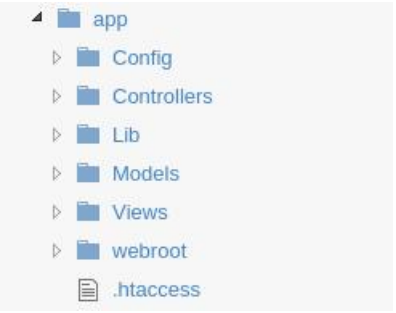

Figura 5.6: Estrutura interna da pasta app

5.4.1 Config

Pasta destinada à configuração da aplicação. O projeto inicia apenas com uma classe dentro desta pasta que é a *DATABASE_CONFIG*, responsável pela configuração da conexão com o banco de dados da aplicação. Esta classe segue a seguinte estrutura:

```
class DATABASE_CONFIG {
    public static $default = array(
        'host' => '',
        'login' => '',
        'senha' => '',
        'banco' => '',
    );
}
```

A configuração da conexão é feita através do atributo **$default**, que é iniciado como um vetor e cada índice recebe uma string referente aos dados de acesso.

5.4.2 Model

A pasta Model guarda todas as classes de comunicação com o banco de dados, separadas por sessão. Para cada parte do sistema, pode-se criar uma nova classe de modelo para deixar a aplicação mais separada e de fácil entendimento. A class de modelo segue um padrão para a criação, mostrado no exemplo abaixo.

```
class Exemplo_Model extends Model {
    public $_tabela = 'exemplo';
}
```

Esta é a forma mais básica para a criação da classe de um modelo *"exemplo"*. A classe passa a ter todas as funcionalidades da classe *Model* pela herança, porém direcionada à tabela *"exemplo"*.

Seguindo as boas práticas de programação e respeitando o *MVC*, todas as consultas referentes à tabela *"exemplo"* passam a ser feitas nesse modelo.

5.4.3 Controller

O Controller define todas as regras de negócio da aplicação, em cada action específica e podendo ter relações entre si. O controller permite a criação de actions, que é parte do link da aplicação e também métodos comuns, que auxiliam na programação. O que diferencia a action de um método comum é a forma em que é escrita. Vide modelo abaixo de um controller simples.

```
class home extends Controller {

    /**
     * Exemplo de action
     */
    public function index_action($params = null) {
        // view a ser usada
        $this->view('index');
    }

    /**
     * Exempo de método comum
     */
    private function metodo_exemplo() {
        // código
    }
}
```

A Classe de controller estende à classe principal *Controller*, obtendo todos os métodos e atributos desta classe. Um desses métodos é o método *view()*, onde é definido o templeta de view a ser usado.

Para se criar uma *action* é obrigatória a utilização de um método público, a passagem de um vetor como parâmetro e a chamada de uma view dentro do método. Esses fatores que diferenciam a action de um método comum.

5.4.4 View

A View possui uma particularidade na estrutura de suas pastas, que é a seguinte: Para cada controller é criado uma pasta dentro da pasta View, com o mesmo nome do controller, porém com a primeira letra em maiúscula como mostrado no seguinte exemplo:

Para um controller **exemplo**, é criada a pasta **View/Exemplo**, onde dentro de **Exemplo** serão criados todos os arquivos de *views* referente a esse controller. Esses arquivos precisam ser criados com a extensão **.phtml** para serem compreendidos pelo *Framework*.

Dentro de **View** existe a pasta **Layouts**, responsável por todos os templates do projeto.

5.4.5 Lib

Está pasta tem a função de agrupar classes ou bibliotecas específicas para determinado projeto. O acesso a essas classes fica facilitado com a utilização de uma constantes do framework denominada **LIB**.

5.4.6 webroot

É a pasta que passa a ser tratada como a raiz do projeto pelo servidor; nela se encontram a pasta **CSS** com todos os arquivos de estilo do projeto, a pasta **JS** com os arquivos de script e a pasta **IMG** com todas as imagens. Outras pastas podem ser criadas sem restrição.

Além dessas pastas, dois arquivos muito importantes para o *Framework* estão alocadas em webroot, e são esses: **.htaccess** e **index.php**. Esses são os primeiros arquivos a serem lidos pelo *Framework* em qualquer chamada da aplicação. Abaixo pode ser ver o que eles fazem.

- **.htaccess**: É o primeiro arquivo a ser lido, e através dele ocorre o direcionamento da url para a criação da rota a aplicação. Sua funcionalidade é recuperar a *URL* em questão e passar via GET para o arquivo **index.php**. O arquivo segue a estrutura descrita abaixo.

```
<IfModule mod_rewrite.c>
  RewriteEngine on
  RewriteCond %{SCRIPT_FILENAME} !-f
  RewriteCond %{SCRIPT_FILENAME} !-d
  RewriteRule ^(.*)$ index.php?url=$1
</IfModule>
```

- **index.php**: Arquivo responsável pelo iniciar definir constantes, fazer o autoload das classes e iniciar a aplicação. Abaixo alguns pontos importantes deste arquivo.

```
error_reporting(0);
session_start();

// Define o horário do servidor
date_default_timezone_set("Brazil/East");
```

Omite a exibição de erros e define o fuso horário do Brasil para o servidor.

```
$_SERVER['PHP_SELF'] = str_replace("app/webroot/index.php", "",
   $_SERVER['PHP_SELF']);
define('URL', $_SERVER['PHP_SELF']);
define('IMG', URL . "img/");
define('JS', URL . "js/");
define('CSS', URL . "css/");
define('FULL_URL', "http://" . $_SERVER['SERVER_NAME'] . URL);
define( 'LIB', '../../app/Lib/' );
define( 'FILES', '../../app/webroot/files/' );
define( 'CONTROLLERS', '../../app/Controllers/' );
define( 'VIEWS', '../../app/Views/' );
define( 'LAYOUT', '../../app/Views/Layouts/' );
define( 'MODELS', '../../app/Models/' );
define( 'HELPERS', '../../system/helpers/' );
define( 'SYSTEM', '../../system/' );
define( 'CONFIG', '../../app/Config/' );
```

Cria todas as constantes auxliares para o projeto.

```
require_once('../../system/System.php');
require_once('../../system/Controller.php');
require_once('../../system/Model.php');
require_once('../../system/TemplateParser.php');
```

Carrega os arquivos necessários.

```
$start = new System;
$start->init();
$start->run();
```

Por fim, inicia o sistema, rodando a aplicação.

6 *Conclusão*

Tendo como foco principal a criação do *Framework Lothus{PHP}* de acordo com a necessidade de criar uma forma agil de se criar sistemas, o projeto cumpriu com os objetivos esperados, visto que, foram comprovadas as funcionalidades especificadas e validados os objetivos específicos.

O objetivo geral do trabalho foi desenvolver um *Framework* utilizando *PHP* como principal linguagem de programação, e cujo o foco é facilitar a criação de sites e sistema fornecendo bibliotecas capazes de cuidar de toda a parte genéria de um projeto. Para tornar a proposta possível foram implementadas todas as tecnologinas aqui descritas, viabilizando de maneira eficiente a construçã e implantação do *Framework*, atendendo ao que foi proposto.

Referências

[1] CakePHP. **Documentação CakePHP Cookbook 2.x.** Disponível em: http://book.cakephp.org/2.0/pt/index.html. Acesso em: 02 Fev. Maio. 2015.

[2] Linha de Código. **CRUD com JSP.** Disponível em: http://www.linhadecodigo.com.br/artigo/2997/crud-com-jsp.aspx. Acesso em: 25 Abril. 2015.

[3] Node BR. **Aplicações web real-time com Node.js.** Disponível em: http://nodebr.com/. Acesso em: 26 Abril. 2015.

[4] Gerencie você mesmo. **Rsync: Para que serve e como usar.** Disponível em: http://gerencievocemesmo.com.br/site/?p=508. Acesso em: 29 Abril. 2015.

[5] Viva o Linux. **Transferindo arquivos com o rsync.** Disponível em: http://www.vivaolinux.com.br/artigo/Transferindo-arquivos-com-o-rsync. Acesso em: 29 Abril. 2015.

[6] Git Scm. **Git Book.** Disponível em: http://git-scm.com/book/pt-br/v1. Acesso em: 03 Maio. 2015.

[7] Locaweb. **PDO - PHP Data Object.** Disponível em: http://wiki.locaweb.com/pt-br/PDO_-_PHP_Data_Object. Acesso em: 08 Maio. 2015.

[8] PHP. **Documentação PHP.** Disponível em: http://php.net. Acesso em: 15 Maio. 2015.