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

MARCOS ROBERTO BERTUOL

PRADO PHP FRAMEWORK: UM ESTUDO EXPERIMENTAL PARA CONTROLE DE ACESSO A SITES

TRABALHO DE DIPLOMAÇÃO

MEDIANEIRA
2011

MARCOS ROBERTO BERTUOL

PRADO PHP FRAMEWORK: UM ESTUDO EXPERIMENTAL PARA CONTROLE DE ACESSO A SITES

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

Orientador: Me. 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
Análise e Desenvolvimento de Sistemas

TERMO DE APROVAÇÃO

PRADO PHP Framework: um estudo experimental para controle de acesso a sites

Por

Marcos Roberto Bertuol

Este Trabalho de Diplomação (TD) foi apresentado às 13:50 h do dia 17 de novembro de 2011 como requisito parcial para a obtenção do título de Tecnólogo no Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas, 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.

Me. Fernando Schütz
UTFPR – *Campus* Medianeira
(Orientador)

Dr. Hermes Irineu Del Monego
UTFPR – *Campus* Medianeira
(Convidado)

Me. Juliano Rodrigo Lamb
UTFPR – *Campus* Medianeira
(Convidado)

Me. Juliano Rodrigo Lamb
UTFPR – *Campus* Medianeira
(Responsável pelas atividades de TCC)

RESUMO

BERTUOL, MARCOS ROBERTO. PRADO PHP FRAMEWORK: UM ESTUDO EXPERIMENTAL PARA CONTROLE DE ACESSO A SITES. 2011. TRABALHO DE CONCLUSÃO DE CURSO (TECNOLOGIA EM ANÁLISE E DESENVOLVIMENTO DE SISTEMAS), UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ. MEDIANEIRA 2011

Com a utilização de *frameworks* o desenvolvedor de sistemas visa diminuir o tempo utilizado no desenvolvimento de um sistema, com menos esforço e mais produtividade. O presente trabalho apresenta um estudo detalhado sobre o *PRADO PHP Framework*, que traz funcionalidades como componentes que executam tarefas comuns em aplicações *Web*, validação de dados, gerenciamento de leiaute, além de proporcionar reusabilidade de código durante o desenvolvimento de um sistema; e uma aplicação desenvolvida utilizando este *framework* como estudo experimental.

LISTA DE SIGLAS

AJAX	*Asynchronous Javascript and XML*
API	*Application Programming Interface*
CGI	*Common Gateway Interface*
CRUD	*Create, Restore, Update, Delete*
DAO	*Data Access Object*
HTML	*HypertText Markup Language*
HTTP	*Hypertext Transfer Protocol*
PERL	*Practical Extraction and Report Language*
PHP	*PHP Hypertext Preprocessor*
PHP/FI	*PHP Hypertext Preprocessor/Forms Interpreter*
PDF	*Portable Document Format*
UML	*Unified Modeling Language*
URL	*Uniform Resource Locator*
XML	*Extensible Markup Language*

LISTA DE FIGURAS

SUMÁRIO

1. INTRODUÇÃO

A produtividade é um objetivo a ser buscado em qualquer ambiente de trabalho, seja no desenvolvimento de *softwares* ou não. Quando no desenvolvimento de software, em sistemas com alto grau de complexidade, essa busca pode se tornar difícil, ao combinar com boa qualidade e organização.

Para Michael Porter (1989), “as empresas precisam encontrar meios de crescer e criar vantagens ao invés de apenas eliminar as desvantagens”. A estratégia tem como objetivo criar uma diferenciação na concorrência, onde não é mais questão de ser melhor naquilo que se faz, e sim, uma questão de ser diferente, e nessa etapa envolver outras maneiras de ser diferente, ter um posicionamento estratégico aprimorado.

Perante empresas concorrentes, a chave do sucesso para um bom desempenho de uma empresa pode ser a produção de serviços de alta qualidade com o mínimo custo possível, alimentando a produtividade. Estimativas de tempo e custos estão diretamente ligadas a esta produtividade.

Atualmente existem tecnologias e ferramentas criadas com o intuito de auxiliar no desenvolvimento de aplicações que buscam essa chave. Especificamente no desenvolvimento de aplicações *Web*, o PRADO PHP Framework pode ser a solução.

O PRADO PHP Framework é uma ferramenta de código aberto para desenvolvimento de aplicações *Web* baseado na linguagem PHP 5. O *framework* tem como prioridade possibilitar o máximo de reusabilidade na programação, ajudando um rápido e organizado desenvolvimento. Ele permite ao programador focar-se no que realmente deve ser desenvolvido: a lógica da aplicação.

O presente documento apresenta um estudo detalhado e uma aplicação como estudo experimental sobre o PRADO PHP Framework.

1.1. OBJETIVO GERAL

Desenvolver, como estudo experimental, um sistema para controle de aplicações e servidores de acesso, utilizando o PRADO PHP Framework.

1.2. OBJETIVOS ESPECÍFICOS

Os objetivos específicos do trabalho são:

- Descrever um estudo bibliográfico sobre a evolução na produtividade no desenvolvimento de aplicações (partindo das bibliotecas de classe até os *frameworks*), apresentando a arquitetura, modelo de programação e funcionamento do PRADO PHP Framework;
- Desenvolver um protótipo, como estudo experimental, utilizando o PRADO PHP Framework;
- Demonstrar a aplicação.

1.3. JUSTIFICATIVA

Diversas tecnologias novas vão surgindo com o intuito de fornecer mais opções e produtividade aos desenvolvedores. Além de novas linguagens, existem também cada vez mais novos *frameworks* para isto.

Os desenvolvedores, principalmente os mais antigos, porém, podem ainda possuir certa resistência a novas tecnologias ou *frameworks*, por diversos motivos: seja por já trabalharem com certa tecnologia e/ou ferramenta há algum tempo, por acreditarem que não necessitam de códigos alheios, ou ainda por desconfiarem destes códigos.

Este trabalho visa apresentar uma nova alternativa para estes desenvolvedores no desenvolvimento de aplicações *Web* com o PHP.

O *framework* em questão foi escolhido por ser completo: possui diversas facilidades na apresentação para o usuário final, bem como uma ampla possibilidade de reusabilidade de códigos.

Segundo Dall'Oglio (2007) ,"o PHP é uma das linguagens mais utilizadas no mundo". Para o autor, a popularidade do PHP se deve à facilidade em criar aplicações dinâmicas com suporte à maioria dos bancos de dados existentes. O PHP tem um conjunto de funções utilizadas por meio de uma estrutura flexível de programação, que permitem desde a criação de simples *sites* até complexas aplicações de negócio.

Segundo McGraw Hill (2007, p. 58), em 2006 havia mais de 20 milhões de *Web* sites utilizando a linguagem PHP. A expectativa deste número crescer ainda

mais se concretizou e, de acordo com o índice da comunidade TIOBE de setembro de 2011 o PHP é hoje a quarta linguagem de programação mais utilizada em todo o mundo.

2. REVISÃO BIBLIOGRÁFICA

Este capítulo apresenta uma revisão bibliográfica dos elementos utilizados para a realização do estudo experimental proposto neste trabalho.

2.1. PRODUTIVIDADE

A relação entre o esforço para se produzir algo e o resultado obtido a partir deste esforço é a definição básica de produtividade. Entende-se que quanto menor o esforço e maior o resultado obtido a partir deste esforço, maior a produtividade alcançada.

No início da programação o desenvolvedor tinha a seu dispor o que se conhece por bibliotecas de classe, que contém código pronto, podendo ser operações matemáticas, por exemplo. Nada mais são do que classes auxiliares que podemos utilizar em qualquer projeto.

Com a popularização da programação, a busca pela produtividade veio à tona, e bibliotecas de classes já não eram mais suficientes. Foram, então, criados os *toolkits*, que em sua essência, ainda são bibliotecas de classe.

Ainda assim, os *toolkits* não eram muito produtivos e os desenvolvedores precisavam de algo melhor. Surgiram então os *frameworks*, com inúmeras vantagens e facilidades para o desenvolvedor de *software*.

> Os *frameworks* facilitam o desenvolvimento de *software*, permitindo que os programadores se ocupem mais com os requerimentos do *software* do que com os detalhes tediosos, de baixo nível do sistema. Com o uso de *frameworks*, os programadores passam a ter o controle de seu tempo e de seus códigos-fonte, as tarefas repetitivas são minimizadas, os projetos são concluídos em menos tempo, os padrões são seguidos, e a programação volta a ser uma arte. (MINETTO, 2007).

Quando um desenvolvedor começa a estudar um novo *framework* ele se depara muitas vezes com uma forma diferente de programar ou até de pensar um sistema. É necessário aprender uma sintaxe diferente, convenções para nomes de arquivos, variáveis e tabelas de banco de dados. Além disso, muitas vezes surge a sensação de estar “engessado”, pois é preciso fazer as coisas da forma que o *framework* trabalha, de modo que qualquer coisa diferente requer um empenho maior. Contudo, as vantagens a médio e longo prazo fazem esse pequeno esforço inicial valer.

2.2. PHP

De acordo com o manual do site oficial, “PHP (um acrônimo recursivo para “PHP: *Hypertext Preprocessor*”) é uma linguagem interpretada de código aberto para uso geral, muito utilizada e especialmente provida para o desenvolvimento de aplicações *Web* embutível dentro do *HTML (HypertText Markup Language)*”.

2.2.1. História do PHP

Páginas estáticas que chamavam outras páginas estáticas e que dificilmente eram atualizadas marcaram o início da *Web*. Com o passar do tempo diversas linguagens foram desenvolvidas para dinamizar e interagir mais com o público alvo. O PHP foi uma das que mais obteve sucesso.

Para Soares (2005), “PHP é uma linguagem de criação de scripts embutida em HTML no servidor”.

Segundo Thomson & Welling (2005), “a linguagem foi concebida em 1994 como resultado do trabalho de uma única pessoa, Rasmus Lerdorf”. Ainda segundo Thomson & Welling (2005), “o PHP foi adotado por outras pessoas inteligentes e foi reescrito três vezes para proporcionar o amplo e aperfeiçoado produto que vemos hoje”.

Para Melo & Nascimento (2007), da mesma forma que aconteceu com o sistema operacional Linux, diversos entusiastas de software livre tomaram conhecimento do novo pacote PHP/FI, que fora lançado em 2007, e então passaram a contribuir com o aprimoramento do mesmo.

O PHP 3.0 foi oficialmente lançado em Junho de 1998, depois de ter passado aproximadamente nove meses em testes públicos. Os objetivos do projeto eram melhorar o desempenho de aplicações complexas, e melhorar a modularidade do código base do PHP. Tais aplicações foram possíveis por causa das novas características do PHP 3.0 e o suporte a uma variedade de banco de dados de terceiros e APIs (*Application Programming Interface)*. O PHP 3.0 não foi projetado para trabalhar com aplicações muito complexas eficientemente. (A História do PHP, 2011).

A versão 4.0 do PHP, com uma série de novas características, foi oficialmente lançada em Maio de 2000, quase dois anos após o seu predecessor, o PHP 3.0. (A História do PHP, 2011).

A atual versão, o PHP 5 foi lançado em julho de 2004 depois de um longo desenvolvimento e vários *pre-releases*. Ele introduziu principalmente o core, a Zend Engine 2.0 com um novo modelo de orientação a objetos, além algumas outras características. (A História do PHP, 2011).

2.2.2. Características e Vantagens

Segundo Converse (2001, p. 67), "... o PHP destaca-se pela sua capacidade, confiabilidade e facilidade de uso. Além disso, oferece o melhor tipo de conectividade para todos os servidores *back-end*".

Segue algumas das principais características do PHP, características estas que influenciaram e contribuíram diretamente para a popularidade do mesmo:

- Tendo como foco específico o desenvolvimento para *Web*, a linguagem contém inúmeros recursos nativos, sem que haja a necessidade da instalação de bibliotecas adicionais, como: conexão com banco de dados, manipulação de arquivos, manipulação de documentos específicos como XML (*Extensible Markup Language*) e PDF (*Portable Document Format)*;
- Tendo uma sintaxe clara e simples, ele torna-se de fácil aprendizado e ao mesmo tempo bastante poderoso. O código é de fácil leitura e entendimento, estimulando um rápido desenvolvimento das aplicações;
- É gratuito e multi-plataforma.

Além destas características citadas, existem algumas vantagens que devem ser mencionadas:

- Extremamente rápida: como seu código é embutido no HTML, o tempo para processar e carregar a página é muito rápido quando comparado a outras linguagens;
- Suporte: a começar pela documentação oficial, que é bastante completa, passando pelas diversas publicações de livros técnicos até chegar a enorme comunidade de programação espalhada pelo mundo inteiro;
- Segurança: o código PHP programado é invisível para o usuário final;
- Orientação a objetos: a partir da versão 5 o PHP traz nativamente todas as características da orientação a objetos.

2.3. FRAMEWORKS

Segundo Oxford (Oxford *Dictionary*, 200, p.535), *framework* significa parte de uma construção ou objeto que suporta sua carga e dá a ele uma forma; um conjunto de ideias, crenças e regras que são usadas como base para julgar; a estrutura de algum sistema.

Um *framework* possui um conjunto de ferramentas que realizam várias tarefas de baixo nível que a grande maioria das aplicações precisam. Algumas destas tarefas, além de necessárias, são também repetitivas em alguns sistemas e, ao invés de escrever ou reescrever estes códigos, usa-se os componentes e facilidades prontas que o *framework* vem a fornecer para o desenvolvedor.

Entretanto, até mesmo o melhor *framework* pode não conter todas as funcionalidades desejadas. Por isso é preciso que ele seja flexível o suficiente para permitir que o desenvolvedor crie ou personalize as funcionalidades que bem entender.

2.3.1. Framework para Web

Quando se trata de *framework* no contexto de desenvolvimento *Web*, trata-se basicamente de um conjunto de ferramentas que são destinadas a ajudar o desenvolvedor com as tarefas comuns, que se tornam tediosas e repetitivas. Um bom *Web framework* tem a função de amenizar essas tarefas. Ele poderá fornecer bibliotecas para tratar de diversas operações, como modelagem de dados, acesso ao banco, *templates* de páginas, manipulação de sessão, mapeamento de URL (*Uniform Resource Locator*) e vários outros componentes. Como mencionado, o ideal é que estes sejam desenvolvidos de uma forma genérica para que seja possível adaptá-los as necessidades do programador de acordo com a aplicação.

2.3.2. Considerações

Há ainda certa resistência quanto à utilização de *frameworks*, principalmente no contexto *Web*. Alguns desenvolvedores julgam que não se faz necessário a utilização do mesmo, já que eles podem criar suas próprias ferramentas de acordo com suas necessidades, sem a necessidade de utilizar códigos de um terceiro.

Apesar de ser um argumento válido, deve-se destacar o poder de um *framework* na busca pela produtividade, e não ignorar as vantagens da conveniência.

Pode existir também por parte de alguns desenvolvedores a desconfiança sobre um código elaborado por terceiros. Essa desconfiança não é justificável, visto que a própria linguagem, no caso o PHP, teve seu interpretador escrita por terceiros, como fundamentado anteriormente.

É verdade que algumas das funções de um *framework* podem não ser utilizadas no desenvolvimento de um projeto. Deve-se levar em conta que em outros, porém, elas poderão vir a ser utilizadas e, mesmo que não sejam, este excesso de códigos não deverá influenciar em nada o desempenho da aplicação. Há de se entender que *frameworks* são bibliotecas de códigos que tentam satisfazer propósitos gerais. Um bom exemplo é o próprio PHP que conta com inúmeras funções que não são utilizadas em determinado projeto. Os *frameworks* atuais ainda contam com uma forma de modularização bastante eficiente, onde o desenvolvedor poderá desmembrar funcionalidades as quais não utilizará em seu projeto.

2.4. PRADO PHP FRAMEWORK

"PRADO é um *framework* baseado em componentes e orientado a eventos para o rápido desenvolvimento de aplicações *Web* na linguagem PHP 5." (*What is* PRADO?, 2011).

Ele reconceitualiza o desenvolvimento de aplicações *Web* em termos de componentes, eventos e propriedades ao invés de procedimentos, URLs e parâmetros.

A inspiração original do PRADO veio do *Apache Tapestry*, um *Web framework* para Java orientado a componentes. Durante o *design* e a implementação do *framework*, muitas ideias foram abstraídas do Borland Delphi e do Microsoft ASP.NET. Quem já tem alguma afinidade com estas linguagens poderá ter um melhor entendimento sobre o PRADO.

Sua primeira versão veio em junho de 2004 e fora escrita em PHP 4, por Qiang Xue, fundador e desenvolvedor do mesmo. Mais tarde seria reescrito em PHP 5, o que foi considerado uma jogada inteligente pelo próprio autor do *framework*, que justificou com o novo modelo de objetos provido pela nova versão da linguagem. Em agosto do mesmo ano o projeto foi liberado como um projeto de código aberto. Com

o suporte da equipe de desenvolvimento e dos usuários do PRADO, ele evoluiu para a versão 2.0 em meados de 2005.

Em maio de 2005, os desenvolvedores decidiram reescrever completamente o *framework*, a fim de resolver algumas questões fundamentais encontradas nesta versão 2.0 e abstrair mais algumas do Microsoft ASP.NET 2.0. Depois de quase um ano e muito trabalho, com mais de 50 mil novas linhas de código, a versão 3.0 finalmente estava pronta, em abril de 2006.

A partir da versão 3.0, esforços significativos foram alocados para assegurar a qualidade e estabilidade do PRADO. Se dissermos que as versões 2.x e 1.x do *framework* não são a prova de conceito de trabalho, podemos dizer que a versão 3.x cresceu para um projeto sério que é adequado para o desenvolvimento de aplicações de negócio.

Segundo Qiang Xue (2011), o primeiro objetivo do *framework* é permitir o máximo de reusabilidade na programação *Web*. Por reusabilidade entende-se não apenas reusar o seu próprio código, mas também o código de outras pessoas de uma maneira fácil. Este princípio diminui dramaticamente o tempo de desenvolvimento de uma aplicação. Ainda segundo Qiang Xue, a introdução do conceito de componentes visa este propósito.

Para alcançar este objetivo, o PRADO estipula um protocolo de escrita e uso de componentes para construir aplicações *Web*. Um componente é uma unidade de *software* que interage com si mesmo e pode ser reutilizado em customizações triviais. Novos componentes podem ser criados pela simples composição de componentes já existentes.

Para facilitar a interatividade com componentes, o PRADO implementa um paradigma de programação orientada a eventos que habilita a delegação de comportamentos extensíveis à componentes. Atividades do usuário final, como clicar num botão de envio, são capturadas como eventos do servidor. Métodos ou funções podem ser disparados por estes eventos e, quando o evento acontece, eles são invocados automaticamente para responder. Comparando com a programação *Web* tradicional, onde desenvolvedor têm de lidar com as variáveis *POST* ou *GET*, a programação orientada a eventos ajuda o desenvolvedor a focar-se melhor na lógica necessária e reduz significativamente o código repetitivo de baixo nível.

Em resumo, desenvolver uma aplicação *Web* com o PRADO envolve principalmente instanciar tipos de componentes pré-construidos, configurá-los definindo suas propriedades, responder a seus eventos escrevendo funções para manipulá-los e compor os mesmos em páginas para a aplicação.

O PRADO é comumente citado como um *framework* único. De fato, ele é tão único que pode transformar a programação PHP em uma tarefa menos tediosa. A lista a seguir é um sumário com as principais características do *framework*: (*What is* PRADO?, 2011).

- Reusabilidade: códigos seguindo o protocolo de componentes do PRADO são altamente reusáveis. Isso beneficia as equipes de desenvolvimento em longo prazo, assim eles podem reutilizar seus próprios trabalhos anteriores e integrar o trabalho de outros facilmente;

- Programação orientada a eventos: atividades do usuário final, como clicar em um botão de envio, são capturadas como eventos do servidor para que o desenvolvedor tenha um foco melhor em lidar com interações do usuário;

- Integração da equipe: apresentação e lógica são armazenadas separadamente. Aplicações do PRADO são temáticas;

- Controles *Web* poderosos: PRADO vem com um conjunto de componentes poderosos para lidar com interfaces *Web*. Páginas *Web* altamente interativas podem ser criadas com algumas linhas de código. Por exemplo, usando o componente *datagrid*, pode-se criar rapidamente uma página que apresenta uma tabela de dados, contendo paginação, classificação, edição e exclusão de linhas desta tabela;

- Forte suporte a banco de dados: desde a versão 3.1, o PRADO foi equipado com um suporte a banco de dados bastante completo, que é nativamente escrito e, portanto, se encaixa perfeitamente com a parte restante do *framework*;

- Suporte a AJAX (*Asynchronous Javascript and XML)*: utilizar AJAX no PRADO nunca foi tão fácil com o inovador conceito de controles ativos, introduzidos na versão 3.1. O desenvolvedor pode facilmente escrever uma aplicação utilizando AJAX sem escrever uma única linha de código *javascript*. Na verdade, utilizar os controles ativos não é muito diferente de utilizar os controles não-AJAX;

- Suporte a I18N e L10N: o PRADO inclui suporte completo para construir aplicações com múltiplas linguagens;

- Espaço para trabalhos existentes: PRADO é um *framework* genérico que foca a camada de apresentação. Ele não exclui a possibilidade para os desenvolvedores de utilizar a maioria das bibliotecas de classe ou *toolkits*;
- Outras características: poderosa manipulação de erros e exceções; tratamento de erro personalizável; autenticação e autorização extensíveis; medidas de segurança como prevenção de *cross-site script*, proteção de *cookies*, etc.

Para melhor explicar como o *framework* funciona, sua arquitetura, configurações, componentes, controles, páginas, módulos e serviços devem ser apresentados. As subseções deste capítulo apresentam tais conceitos.

2.4.1. Arquitetura

O PRADO é primeiramente um *framework* de apresentação, embora ele não se limite a isto. O *framework* foca na programação *Web*, onde trata a maior parte do tempo com interações do usuário, que será baseada em componentes e orientada a eventos para que os desenvolvedores possam ser mais produtivos. A árvore de classes é apresentada na Figura 1 e mostra algumas das mais importantes classes fornecidas pelo PRADO.

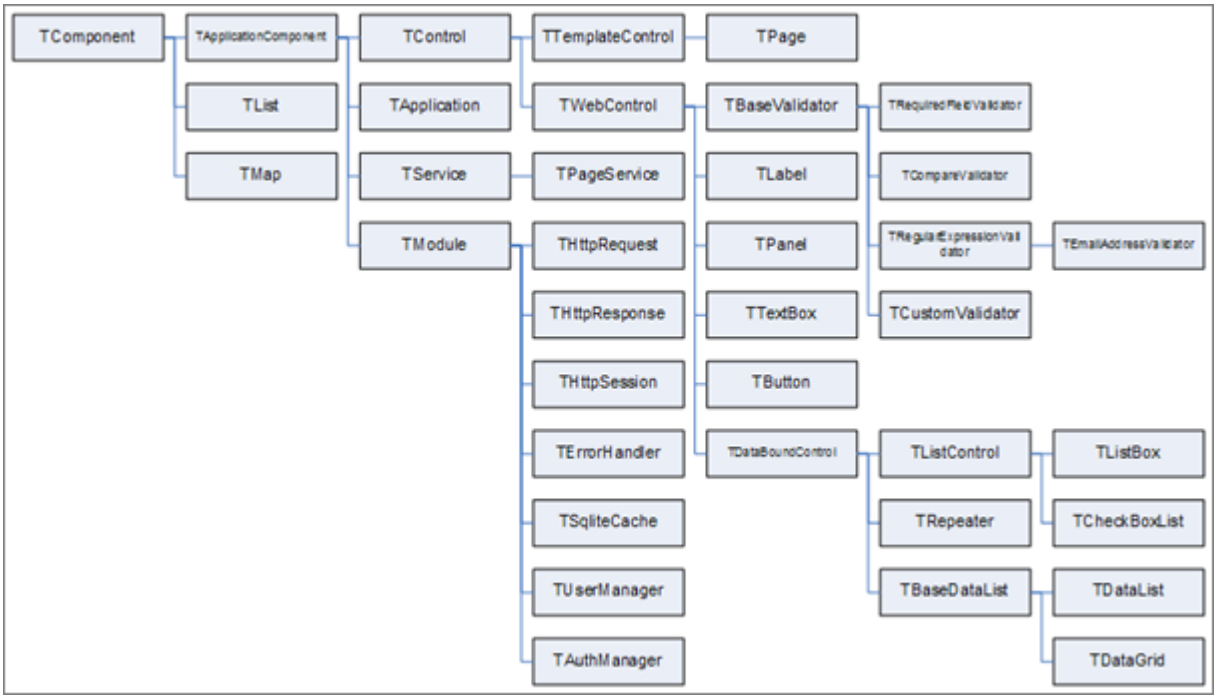

Figura 1 - Árvore com as principais classes fornecidas pelo PRADO
Fonte: Adaptado de Pradosoft (2011).

O diagrama exibido na Figura 1 deixa claro que todas as classes do *framework* estendem a classe `TComponent`. Fazendo uma analogia com a linguagem Java, esta classe PHP seria como a classe `Object` do mesmo. É possível também a visualização de alguns componentes prontos para utilização na parte *Web*, como o componente `TDataGrid`, que representa uma tabela para visualização de dados.

2.4.2. Aplicação

Uma aplicação é uma instância da classe `TApplication`, ou uma classe derivada desta. Esta classe irá gerenciar os módulos que fornecerem diferentes funcionalidades para a aplicação. Ela irá também fornecer os serviços para o usuário final. Além disso, é aqui que são armazenados diversos parâmetros utilizados na aplicação. Em uma aplicação PRADO, a instância da aplicação é o único objeto que é acessível globalmente.

Uma aplicação PRADO básica contém, no mínimo, dois arquivos: um arquivo de entrada e um arquivo de *template* de página. Eles devem estar organizados como na Figura 2.

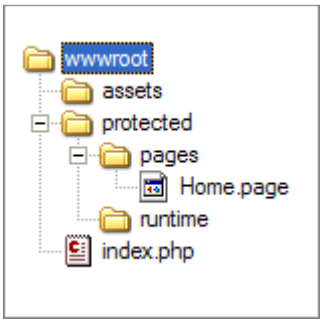

Figura 2 - Estrutura de pastas de uma aplicação PRADO

Segue uma descrição da estrutura apresentada na Figura 2:

- wwwroot - Raiz dos documentos Web;
- index.php - o *script* de entrada de uma aplicação PRADO;
- assets - diretório que contem a publicação dos arquivos privados da aplicação;
- protected - diretório onde ficam armazenados dados da aplicação e arquivos de *script* privados;

- runtime - diretório utilizado pelo PRADO para armazenar informações em tempo de execução;
- pages - diretório onde ficam localizadas todas as páginas da aplicação;
- Home.page - página padrão que é retornada quando não é especificada a página de destino em uma requisição.

2.4.2.1. Ciclos de vida de uma aplicação

Segue, na Figura 3, o ciclo de vida de uma aplicação PRADO.

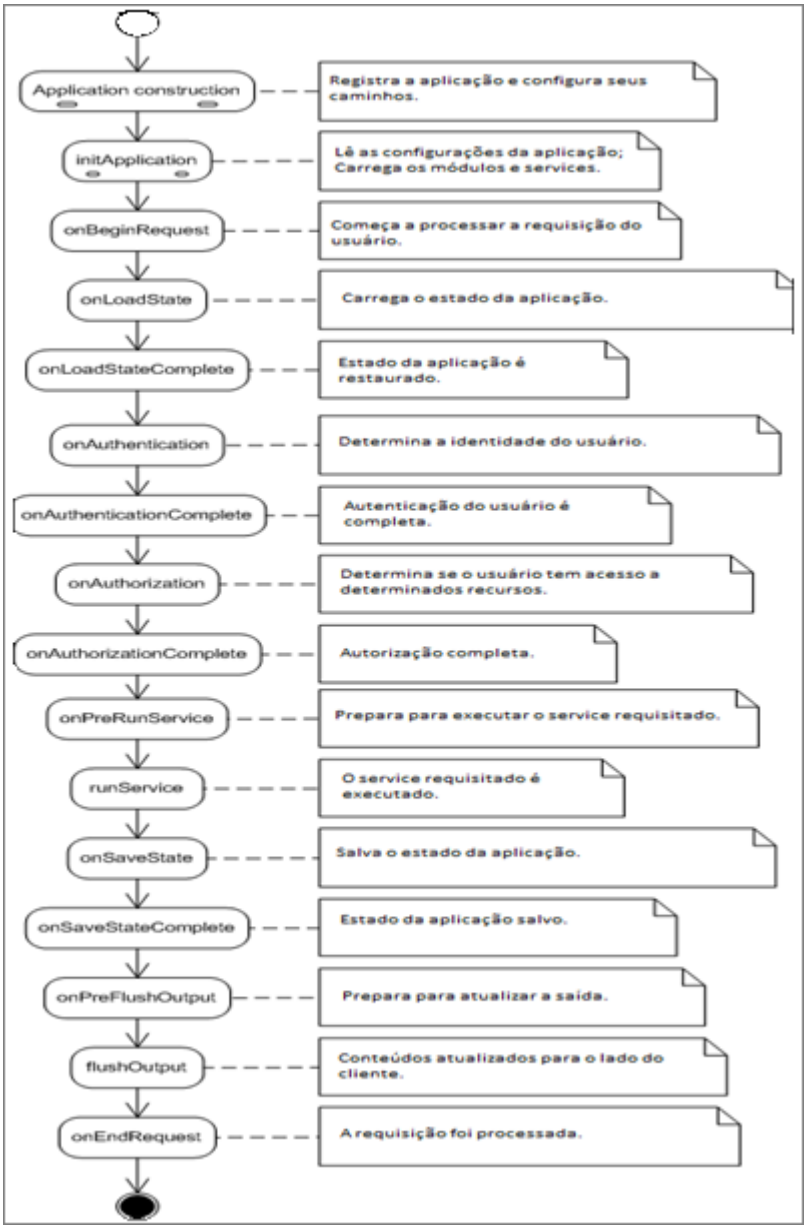

Figura 3 - Ciclo de vida de uma baplicação PRADO
Fonte: Adaptado de Pradosoft (2011).

2.4.2.2. Configurações da aplicação

Configurações da aplicação são utilizadas para especificar comportamentos globais de uma aplicação. Elas incluem especificações de diretórios, *namespaces* utilizados e configurações de módulos, serviços e propriedades.

Estas configurações da aplicação são armazenadas em um arquivo XML chamado `application.xml`, o qual deve estar localizado sobre o diretório base da aplicação. O formato deste arquivo pode ser como o apresentado na Figura 4.

```
1  <?xml version="1.0" encoding="utf-8"?>
2
3  <application id="demo" mode="Debug">
4    <modules>
5      <!-- DECLARAÇÃO DOS MÓDULOS -->
6    </modules>
7
8    <services>
9      <!-- DECLARAÇÃO DOS SERVICES -->
10     <service id="page" class="TPageService" DefaultPage="Home" />
11   </services>
12
13   <!-- DECLARAÇÃO DOS PARÂMETROS
14   <parameters>
15     <parameter id="param1" value="value1" />
16     <parameter id="param2" value="value2" />
17   </parameters>
18   -->
19 </application>
```

Figura 4 - Arquivo de configurações da aplicação

Na Figura 4 pode-se observar que existe um nó principal no XML, que se refere à aplicação em si. Os módulos, serviços e parâmetros são configurados em nós internos deste nó principal.

2.4.3. Controles

Um controle é uma instância da classe `TControl`, ou uma subclasse da mesma. É uma definição de componente que não se limita a interface do usuário.

2.4.3.1. Árvore de controles

Controles são relacionados uns com os outros através da relação pai-filho, cada controle pai pode ter um ou vários controles filhos. A renderização do resultado de um controle filho é, normalmente, utilizada para compor a apresentação de um controle pai. A relação entre controles pais e controles filhos transforma um conjunto de controles em uma árvore de controles. Uma página está no topo desta árvore, a qual a apresentação é retornada para os usuários finais.

Esta relação de pai-filho é, normalmente, estabelecida pelo *framework* através de *templates*. No código, pode-se especificar um controle como filho de outra utilizando um dos métodos descritos na Figura 5.

```
1  $principal->Controls->add($filho);
2  $principal->Controls[]=$filho;
```

Figura 5 - Métodos para especificar a relação de um controle

Na Figura 5 a propriedade *Controls* faz referência à coleção de controles filhos de um controle pai.

2.4.3.2. Identificação de controles

Cada controle possui uma propriedade ID que pode identificar unicamente o mesmo. Além desta, cada controle possui as propriedades `UniqueID` e `ClientID` que podem ser utilizadas para identificar globalmente um controle na árvore a que este controle pertence. São propriedades bastante semelhantes. O primeiro é utilizado pelo *framework* para determinar a localização do controle correspondente na árvore, enquanto o último é utilizado principalmente no lado do cliente como IDs de tags HTML.

2.4.3.3. ViewState e ControlState

HTTP (*Hypertext Transfer Protocol*) é um protocolo *stateless*, o que significa que ele não fornece suporte para interação contínua entre o usuário e o servidor. Cada requisição é considerada independente da outra. Uma aplicação *Web*, entretanto, necessita muitas vezes saber o que um usuário fez em requisições

anteriores. Para isso, desenvolvedores introduzem as sessões para buscar essas informações.

O PRADO buscou os conceitos de *viewstate* e *controlstate* do Microsoft ASP.NET para fornecer esse mecanismo de interação contínua entre usuário e servidor. Um valor armazenado no *viewstate* ou no *controlstate* pode estar disponível nas próximas requisições se as mesmas forem em forma de submissões à mesma página pelo mesmo usuário (os *postbacks*). A diferença entre estes controles é que o *viewstate* pode ser desabilitado, enquanto o *controlstate* não pode.

Estes controles são implementados pela classe `TControl` e são comumente utilizados para definir várias propriedades. Para salvar e obter valores destes controles utiliza-se os métodos descritos na Figura 6.

```
1  $this->getViewState('Nome',$valorPadrao);
2  $this->setViewState('Nome',$valor,$valorPadrao);
3  $this->getControlState('Nome',$valorPadrao);
4  $this->setControlState('Nome',$valor,$valorPadrao);
```

Figura 6 - Manipulando valores do ViewState e do ControlState

Onde `$this` se refere à instância do controle, `'Nome'` a uma chave que identifica a propriedade, `$valor` ao valor persistido nesta chave e `$valorPadrao` é um valor padrão. Este último é opcional. Quando obtendo valores, seja de um viewstate ou de um controlstate, se o valor da chave correspondente não existir, o valor padrão será retornado.

2.4.4. Páginas

Páginas são os controles de nível mais alto, as quais não possuem pai. As apresentações de páginas são diretamente mostradas aos usuários finais, que as acessam enviando requisições de serviços de páginas.

Cada página deve possuir um arquivo de *template*. O nome deste *template* deve conter o sufixo *.page*. O nome do arquivo, sem o sufixo, é o nome da página. O PRADO irá tentar localizar o arquivo da classe da página sob o diretório que contem o arquivo *template* da mesma. Este arquivo da classe da página deve conter o mesmo nome do arquivo *template*, porém com o sufixo *.php*. Se o arquivo da classe não for encontrado, a página utilizará a classe `TPage` por padrão.

2.4.4.1. PostBack

A submissão de um formulário é chamada de *postback* se o envio for feito para a mesma página que contém o formulário. *Postback* pode ser considerado um envio que aconteceu no lado do cliente, invocado pelo próprio. O PRADO irá tentar identificar qual controle no lado do servidor é responsável por tratar este evento *postback*.

2.4.4.2. Ciclos de vida de uma página

Compreender o ciclo de vida de uma página é crucial para entender a programação do PRADO. Estes ciclos de vida se referem à transição de estados de uma página quando servindo esta página para os usuários finais. Estes ciclos de vida estão representados, na Figura 7.

Figura 7 - Ciclo de vida de uma página
Fonte: Adaptado de Pradosoft (2011).

2.4.4.3. Configurações de páginas

Configurações de página são usadas principalmente pelo *service* `TPageService` para modificar ou adicionar configurações da aplicação. Uma configuração de página está associada a um diretório que armazena arquivos referentes a páginas. Estas configurações estão armazenadas em um arquivo XML chamado de `config.xml`. Caminhos, módulos e parâmetros especificados nesses arquivos serão adicionados ou mesclados nas configurações da aplicação. Na Figura 8 pode-se observar o formato de um arquivo de configuração de página.

```
<configuration>
  <paths>
    <alias id="IdAlias" path="CaminhoAlias" />
    <using namespace="Namespace" />
  </paths>
  <modules>
    <module id="IdModulo" class="ClasseModulo" PropertyName="ValorPropriedade" />
  </modules>
  <parameters>
    <parameter id="IdParametro" class="ClasseParametro" PropertyName="ValorPropriedade" />
  </parameters>
  <authorization>
    <allow pages="IdPagina1,IdPagina2" users="Usuario1,Usuario2" roles="Role1,Role2" verb="get" />
    <deny pages="IdPagina1,IdPagina2" users="Usuario1,Usuario2" roles="Role1,Role2" verb="post" />
  </authorization>
  <pages PropertyName="ValorPropriedade">
    <page id="IdPagina" PropertyName="ValorPropriedade" />
  </pages>
</configuration>
```

Figura 8 - Arquivo de configuração de páginas

2.4.5. Módulos

Um módulo é uma instância de uma classe que implementa a interface `IModule`. Ele é comumente desenhado para prover funcionalidades específicas que podem ser plugadas na aplicação PRADO e compartilhadas por todos os componentes desta aplicação.

O PRADO usa configurações para especificar quando carregar um módulo, quais tipos de módulos, e como inicializar os módulos carregados. Desenvolvedores podem substituir os módulos do núcleo do *framework* com suas próprias implementações através das configurações da aplicação, ou também escrever novos módulos para prover funcionalidades adicionais. Um módulo pode, por exemplo, ser desenvolvido para prover a lógica comum de uma base de dados para uma ou várias páginas.

Existem três módulos do núcleo do *framework* que são carregados por padrão quando a aplicação é executada. São eles: *request*, *response*, e o *error handler*. Em adição, o módulo *session* é carregado quando um usuário utiliza uma sessão na aplicação. O PRADO possui implementações padrão para todos estes módulos, porém, módulos personalizados podem ser configurados ou desenvolvidos para substituir esses módulos já existentes.

2.4.5.1. Módulo Request

O módulo *request* fornece o armazenamento e o esquema de acessos para requisições de usuários enviadas por HTTP. Requisições de usuários vêm de diversas fontes, incluindo URL, dados da sessão, dados de *cookies*, etc. Estes dados podem ser acessados pelo módulo *request*. Por padrão, o PRADO utiliza a classe `THttpRequest` como seu módulo *request*. Este módulo pode ser acessado através da propriedade `Request` da aplicação e de seus controles.

2.4.5.2. Módulo Response

O módulo *response* programa o mecanismo para controle da saída para os usuários. Ele pode ser configurado para controlar como a saída será armazenada no lado do cliente. Pode, também, ser utilizado para enviar *cookies* de volta ao mesmo. Por padrão, o PRADO utiliza a classe `THttpResponse` como seu módulo *response*. Ele pode ser acessado através da propriedade `Response` da aplicação e de seus controles.

2.4.5.3. Módulo Session

O módulo *session* encapsula as funcionalidades relacionadas a manipulação da sessão do usuário. Ele é automaticamente carregado quando a aplicação utiliza sessões. Por padrão, o PRADO utiliza a classe `THttpSession` como módulo *session*, que é simplesmente uma cópia das funções de sessão providas pelo PHP. Este módulo pode ser acessado através da propriedade `Session` da aplicação e de seus controles.

2.4.5.4. Módulo Error Handler

O módulo para manipulação de erros é utilizado para capturar e processar todas as condições de erro em uma aplicação. O PRADO, por padrão, utiliza a classe `TErrorHandler` para este fim. Ele captura todos os avisos, notificações e exceções do PHP e mostra de uma forma adequada para os usuários finais. Ele pode ser acessado através da propriedade `ErrorHandler` de uma instância da aplicação.

2.4.6. Services

Um *service* é uma instância de uma classe que implementa a interface `IService`. Cada tipo de *service* processa um tipo específico de requisição de usuários. Por exemplo, o *page service* responde a requisições de usuários para páginas do PRADO.

Um serviço é identificado unicamente pela sua propriedade ID. Por padrão, quando a classe `THttpRequest` é utilizada como módulo *request*, nomes de variáveis *GET* são utilizados para identificar quais *services* um usuário está requisitando. Se uma dessas variáveis for igual ao ID de algum *service*, a requisição é atribuída para este *service*, e o valor da variável *GET* é passado como parâmetro para o mesmo. Para um *page service*, o nome da variável *GET* deve ser *page*. Por exemplo, a URL apresentada na Figura 9 faz uma requisição para a página `Usuarios.Registrar`.

```
http://hostname/index.php?page=Usuarios.Registrar
```

Figura 9 - Requisição de página

Desenvolvedores podem ainda programar *services* adicionais para suas aplicações, os quais servem para processar as requisições dos usuários.

2.4.7. Componentes

Um componente é uma instância da classe `TComponent`, ou uma classe filha da mesma. A classe base `TComponent` implementa o mecanismo de propriedades de componentes e eventos.

2.4.7.1. Propriedades de componentes

Uma propriedade de um componente pode ser vista como uma variável pública que descreve um aspecto específico do componente, como a cor de fundo, tamanho da fonte, etc. Uma propriedade é definida pela existência de um método *getter* e/ou *setter* na classe do componente. Por exemplo, na classe `TControl`, define-se e obtem-se o valor sua propriedade ID utilizando os seus respectivos métodos *getter* e setter, como visto na Figura 10.

```
class TControl extends TComponent {
    public function getID() {
        ...
    }
    public function setID($value) {
        ...
    }
}
```

Figura 10 - Getter e setter da propriedade ID

Para obter ou definir o valor da propriedade ID utiliza-se o código apresentado na Figura 11, observa-se que é exatamente como trabalhar com uma variável.

```
$id = $componente->ID;
$componente->ID = $id;
```

Figura 11 - Definindo e obtendo valor da propriedade ID

Utilizar como ilustrado na Figura 11 é o equivalente a utilizar o *getter* e o *setter* da propriedade, como mostrado na Figura 12.

```
$id = $componente->getID();
$componente->setID( $id );
```

Figura 12 - Definindo e obtendo valor da propriedade ID através de seus métodos getter e setter

Uma propriedade é somente para leitura se a mesma contiver o método *getter*, mas não contiver o método *setter*. Assim como nomes de métodos no PHP são *case-insensitive*, nomes de propriedades também são. Uma classe de um componente herda todas as propriedades de suas classes ancestrais.

2.4.7.2. Subpropriedades

Uma subpropriedade é uma propriedade de alguma propriedade objeto-tipada. Por exemplo, a classe `TWebControl` tem a propriedade `Font` que é do tipo `TFont`. Então a propriedade `Name` da propriedade `Font` se refere a uma subpropriedade.

Para obter ou definir uma subpropriedade, utiliza-se como ilustrado na Figura 13.

```
$name = $component->getSubProperty('Font.Name');
$component->setSubProperty('Font.Name', $name);
```

Figura 13 - Obtendo e definindo o valor de uma subpropriedade

Que é o equivalente a obter e definir a subpropriedade através de seus métodos *getter* e *setter*, como se pode ver na Figura 14.

```
$name = $component->getFont()->getName();
$component->getFont()->setName( $name );
```

Figura 14 - Obtendo e definindo o valor de uma subpropriedade através dos métodos getter e setter

2.4.7.3. Eventos de um componente

Eventos de um componente são propriedades especiais que levam nomes de métodos como seus valores. Anexando (configurando) um método a um evento irá ligar o método aos locais em que o evento é gerado. Portanto, o comportamento de um componente pode ser modificado de uma maneira que não pode ser prevista durante o desenvolvimento de um componente.

Um evento de um componente é definido pela existência de um método cujo nome se inicia com a palavra *on*. O nome do evento é o nome do método e, portanto, é *case-insensitive*. Na Figura 15 ilustra-se o evento `onClick` da classe do componente `TButton`.

```
class TButton extends TWebControl {
    public function onClick( $param ) {
        ...
    }
}
```

Figura 15 - Evento onClick da classe TButton

Isto define um evento chamado `OnClick`, para o qual um manipulador pode ser anexado utilizando uma das maneiras ilustradas na Figura 16, onde `$metodo` referencia a uma chamada válida do PHP (o nome de uma função, por exemplo).

```
1 $botao->OnClick = $metodo;
2 $botao->OnClick->add( $metodo );
3 $botao->OnClick[] = $metodo;
4 $botao->attachEventHandler( 'OnClick' , $metodo );
```

Figura 16 - Configurando um método manipulador para um evento

2.4.7.4. Namespaces

Um *namespace* refere-se a um agrupamento lógico de alguns nomes de classes para que elas possam diferenciar-se de outros nomes de classes, mesmo que seus nomes sejam os mesmos. Um *namespace* do PRADO é basicamente como um pacote do Java. Além deste recurso, as classes do PRADO são prefixadas pela letra 'T' (vêm de *Type*) e, por convenção do *framework*, classes criadas pelo usuário não devem utilizar este prefixo. Os usuários podem, porém, utilizar qualquer outra letra como prefixo.

Um *namespace* do PRADO é considerado como um diretório contendo um ou mais arquivos de classes. Uma classe pode ser especificada sem ambigüidade utilizando um *namespace* seguido pelo nome da classe. Cada *namespace* no PRADO é especificado no formato `'Caminho.Dir1.Dir2'` onde, `'Caminho'` é um pseudônimo de algum diretório, enquanto `'Dir1'` e `'Dir2'` são subdiretórios dentro deste diretório principal. Uma classe chamada `'MinhaClasse'` definida sobre o diretório `'Dir2'` pode ser qualificada como `'Caminho.Dir1.Dir2.MinhaClasse'`.

Para utilizar um namespace no código, faz-se como ilustrado na Figura 17.

```
1  Prado::using('Caminho.Dir1.Dir2.*');
```

Figura 17 - Declaração de um namespace

O código mostrado na Figura 17 acrescenta o diretório referenciado por `'Caminho.Dir1.Dir2'` na lista de diretórios incluídos no PHP para que as classes que estejam contidas neste diretório possam ser instanciadas sem o prefixo do *namespace*. É também possível incluir definições individuais de classes, como demonstrado na Figura 18.

```
1  Prado::using('Caminho.Dir1.Dir2.MinhaClasse');
```

Figura 18 - Declaração de uma classe individual

2.4.7.5. Instanciação de componentes

Instanciação de componentes significa criar instâncias de classes de componentes. Existem duas maneiras de instanciar componentes: instanciação estática e instanciação dinâmica. Os componentes criados são chamados de componentes estáticos e componentes dinâmicos, respectivamente.

2.4.7.5.1. Instanciação de componentes estática

Instanciar componentes estaticamente significa criar componentes através de configurações. O trabalho real da criação é feito pelo *framework*. Por exemplo, em uma configuração da aplicação, pode-se configurar um módulo para ser carregado quando a aplicação for executada. Neste caso, o módulo é um componente estático criado pelo *framework*.

2.4.7.5.2. Instanciação de componentes dinâmica

Instanciar componentes dinamicamente significa criar instâncias de componentes no código PHP. É o mesmo que a criação de objetos comumente utilizada no PHP. Um componente pode ser criado dinamicamente utilizando um dos métodos apresentados na Figura 19.

```
1  $componente = new NomeClasseComponente;
2  $componente = Prado::createComponent('TipoComponente');
```

Figura 19 - Instanciando um componente dinamicamente

Onde `'TipoComponente'` se refere a um nome de uma classe, podendo utilizar o conceito de *namespaces*. A segunda abordagem é introduzida para compensar a falta de apoio a *namespaces* no PHP.

2.4.8. Templates

Templates são utilizados para especificar o leiaute de apresentação de páginas. Um *template* pode conter textos estáticos, componentes ou controles, que contribuem para a apresentação final de uma página.

O formato dos *templates* é basicamente como no HTML, porém com *tags* específicas do PRADO que incluem *tags* de: componentes, controles, comentários, conteúdo dinâmico e propriedades dinâmicas.

2.4.9. Autenticação e Autorização

Como descrito na Figura 3, vários ciclos de vida de uma aplicação são reservados para módulos responsáveis pela autenticação e autorização. Para este propósito, o PRADO fornece o módulo `TAuthManager`, que foi desenhado para ser utilizado em conjunto com o módulo `TUserManager`, que implementa uma base de dados somente leitura de usuários.

Quando ocorre uma requisição de página, o módulo `TAuthManager` irá tentar obter informações do usuário contidas na sessão e, se não encontrar nenhuma, o usuário será considerado como usuário anônimo ou convidado. Para facilitar a identificação do usuário, este módulo fornece dois métodos comumente utilizados: `login()` e `logout()`. Um usuário está autenticado se as entradas de nome do usuário e senha combinarem com um registro na base de dados do usuário, gerenciada pelo módulo `TUserManager`. Por outro lado, um usuário está sem autenticação se esta informação não existir na sessão.

Durante uma requisição de página, o módulo `TAuthManager` irá verificar se aquele usuário tem acesso aquela página, através de algumas regras de autorização, que são baseadas em *roles*. Um usuário tem acesso a uma determinada página se estiver explícito que aquele usuário tem acesso aquela página ou se aquele usuário pertence a uma *role* em particular que, por sua vez, garante acesso aquela página. Se nenhuma destas condições for satisfeita, significa que aquele usuário não pode acessar aquela página e, então, o módulo `TAuthManager` irá redirecionar o usuário para a página de autenticação. Para utilizar estes módulos do PRADO, basta configurá-los nas configurações da aplicação, como demonstrado na Figura 20.

```
<services>
  <!-- DECLARAÇÃO DOS SERVICES -->
  <service id="page" class="TPageService">
      <modules>
          <module id="auth" class="System.Security.TAuthManager"
                    UserManager="users" LoginPage="LoginUsuario" />
          <module id="users" class="System.Security.TUserManager"
                    PasswordMode="Clear">
            <user name="normal" password="normal" />
            <user name="admin" password="admin" />
          </module>
      </modules>
  </service>
</services>
```

Figura 20 - Configuração dos módulos de autenticação e autorização do PRADO

Na Figura 20, a propriedade `UserManager` do módulo `TAuthManager` é configurada para o módulo `'users'`, que neste caso é o `TUserManager`.

Regras de autorização de páginas são especificadas nas configurações de páginas e podem ser tanto para permitir quanto para negar determinada página a determinado grupo de usuários, que é definido por uma *role*. Cada uma dessas regras consiste em quatro propriedades opcionais:

- pages: lista de nomes de páginas, separados por vírgula, as quais esta regra aplicar-se-á. Se vazia ou configurada com um asterisco, esta regra será aplicada para todas as páginas sobre o diretório desta configuração, bem como seus subdiretórios, recursivamente;
- users: lista de nomes de usuários, separados por vírgula, aos quais esta regra aplicar-se-á. Se vazia ou configurada com um asterisco, esta regra será aplicada a todos os usuários, incluindo usuários anônimos e convidados. Um caractere de interrogação ('?') se refere aos usuários anônimos enquanto que o caractere arroba ('@') a todos usuários autenticados;
- roles: lista de nomes de roles, separados por vírgula, as quais esta regra aplicar-se-á. Se vazia ou configurada com um asterisco, a regra irá ser aplicada para todos os papeis;
- verb: método de acesso a página para o qual está regra atenderá, podendo ser configurado como *GET* ou *POST*. Se vazio ou configurado com asterisco, a regra irá ser aplicada para ambos os métodos.

Na Figura 21, pode-se ver um exemplo de utilização destas regras de acesso.

```
13      <authorization>
14          <allow pages="PageID1,PageID2"
15                 users="User1,User2"
16                 roles="Role1" />
17          <deny pages="PageID1,PageID2"
18                 users="?"
19                 verb="post" />
20      </authorization>
```

Figura 21 - Definindo regras de acesso

Quando uma requisição de página está sendo processada, uma lista de regras de autorização estará disponível. Entretanto, apenas a primeira regra efetiva que combinar com o usuário irá processar o resultado da autorização.

2.4.10. Validação de dados

Uma das tarefas mais tediosas e repetitivas no desenvolvimento de um sistema *Web* certamente passa pela validação da entrada de dados nas interfaces do usuário. O PRADO traz alguns componentes prontos que fazem essas validações, a fim de facilitar a vida do desenvolvedor, fazendo com que o mesmo ganhe em produtividade.

Os validadores do PRADO compartilham algumas propriedades em comum, as quais são definidas pela classe base `TBaseValidator` e estão listadas abaixo:

- `ControlToValidate`: especifica o controle de entrada a ser validado. Esta propriedade deve conter o ID do respectivo controle;
- `ErrorMessage`: especifica a mensagem de erro a ser exibida caso os dados não passem pela validação;
- `ValidationGroup`: especifica a qual grupo este validador pertence;
- `EnableClientScript`: especifica quando a validação deve ser executada no lado do cliente;
- `Display`: especifica como a mensagem de erro será exibida. Esta propriedade pode assumir três valores: `None`, `Static` e `Dynamic`;
- `ControlCssClass`: a classe CSS que será aplicada ao controle validado em caso de falha na validação;
- `FocusOnError`: configura o foco para o local da validação, caso a validação falhe.

Alguns dos principais validadores fornecidos pelo PRADO incluem: validador de campo obrigatório, validador de email e validador de tipo de dados.

2.4.10.1. Validador de campo obrigatório

Para validar campos obrigatórios o PRADO fornece o componente `TRequiredFieldValidator`. Ele assegura que o usuário entrou com algum dado em determinado componente de entrada. Na Figura 22, observa-se um exemplo de utilização de um `TRequiredFieldValidator`, o qual irá validar o componente de entrada de dados `TTextBox` com ID igual a '`TextBox1`'.

```
1  <com:TTextBox ID="TextBox1" />
2  <com:TRequiredFieldValidator
3      ValidationGroup="Grupo1"
4      ControlToValidate="TextBox1"
5      ErrorMessage="Campo Obrigatório." />
6  <com:TButton Text="Enviar" ValidationGroup="Grupo1" />
```

Figura 22 - Componente TRequiredFieldValidator

Observa-se que este validador pertence a um grupo chamado '`Grupo1`' e este mesmo grupo está associado ao controle `TButton`, que irá invocar a validação ao ser chamado.

2.4.10.2. Validador de email

Este componente verifica se o texto digitado pelo usuário em um determinado componente de entrada é, potencialmente, um email. Ele utiliza expressões regulares para verificar se o conteúdo fornecido pelo usuário está em um formato válido de email. Esta validação é feita através do componente `TEmailValidator`.

É importante citar que este componente não irá validar se o conteúdo fornecido por usuário for um texto em branco. Para este fim, é necessário utilizar também um `TRequiredFieldValidator`.

2.4.10.3. Validador de tipo de dados

Este componente, por sua vez, irá verificar se a entrada de dados do usuário é do tipo indicado pela propriedade `DataType` do mesmo.

2.4.11. Data Access Object

Nesta seção se demonstra como efetuar uma conexão com um banco de dados, executar comandos SQL e obter valores a partir destes.

Os DAOs (*Data Access Objects*) separam os recursos da interface do cliente dos mecanismos de acesso a dados. Eles separam recursos específicos de acesso a dados da interface genérica do cliente. Como resultado, estes mecanismos de acesso aos dados podem ser alterados independentemente do código que os utiliza.

Os DAOs do PRADO consistem principalmente de quatro classes:

- `TDbConnection`: representa a conexão com um banco de dados;
- `TDbCommand`: representa um comando SQL;
- `TDbDataReader`: classe utilizada para leitura de um resultado de uma consulta;
- `TDbTransction`: representa uma transação do banco de dados.

Para estabelecer uma conexão com um banco de dados, uma instância da classe `TDbConnection` é criada e ativada. É necessário especificar um nome de uma fonte de dados, informação esta que varia dependendo de qual banco de dados será utilizado. Um nome de usuário e uma senha também podem ser passados ao criar uma instância desta classe. Na Figura 23 é criada uma instância desta classe, em seguida é estabelecida a conexão e, por fim, a mesma é fechada.

```
1  $conexao=new TDbConnection($fonteDados,$usuario,$senha);
2  $conexao->Active=true;  // connection é estabelecida
3  $conexao->Active=false;  // connection é fechada
```

Figura 23 - Criando, estabelecendo e fechando uma conexão

Segue abaixo uma lista das fontes de dados mais utilizadas:

- *MySQL*: `mysql:host=localhost;dbname=teste`
- *SQLite*:`sqlite:/path/to/dbfile`
- *ODBC*: `odbc:exemplo`

Uma vez criada e ativada uma conexão com o banco de dados, comandos SQL podem ser executados através da classe `TDbCommand`. Para criar um desses comandos, utiliza-se o método `createCommand()` da classe `TDbConnection`,

passando como parâmetro o comando SQL desejado. Este comando pode ser executado através de dois métodos:

- `execute`(): este método irá executar uma SQL qualquer que não tenha retorno, tais como um comando `insert`, `update` ou um comando `delete`. Caso obtenha sucesso, ele retornará o número de linhas afetadas pela sua execução;

- `query()`: este método, por sua vez, é utilizado para comandos que tenham algum tipo de retorno, como em um `select`. Caso obtenha sucesso, ele retornará uma instância da classe `TDbDataReader`, com o resultado do comando executado.

Após executada uma consulta com algum retorno através do método `query()`, obtem-se uma instância da classe `TDbDataReader`, a qual deverá percorrer para obter seu(s) valor(es). Para isso, utiliza-se o método `read()` desta classe. Fica a critério do desenvolvedor utilizar quaisquer um dos métodos de iteração existentes no PHP para este fim.

2.4.12. Ferramenta de linha de comando

Esta ferramenta do PRADO fornece uma solução para uma das tarefas mais tediosas quando se inicia um novo projeto: a criação do esqueleto do sistema. Quem fornece esta opção é o *script* PHP `prado-cli.php`, localizado na pasta principal do *framework*.

Para utilizá-la, é necessário que comandos PHP possam ser executados a partir do terminal.

Criar esse esqueleto da aplicação torna-se uma tarefa bastante fácil com a utilização desta ferramenta, é necessário apenas a utilização de um comando, o qual está ilustrado na Figura 24.

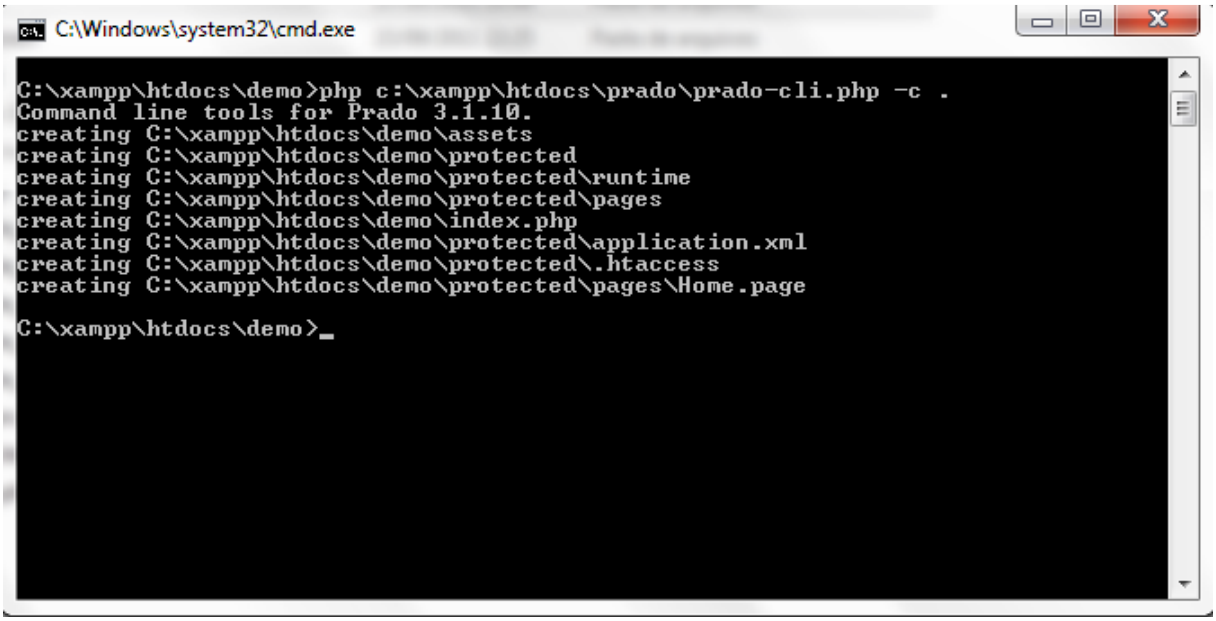

Figura 24 - Utilizando o script de linha de comando do PRADO

Nesta figura, estando dentro da pasta 'demo' efetua-se a execução do *script* PHP `prado-cli.php`, informando o caminho completo do mesmo.

.

3. MATERIAL E MÉTODOS

O estudo teórico realizado sobre o PRADO PHP Framework fornece uma visão mais abstrata desta ferramenta. Porém, como o objetivo deste trabalho é demonstrar a produtividade utilizando o *framework*, é importante colocá-lo em prática e, através do desenvolvimento de um sistema, é possível notar com clareza as dificuldades e facilidades encontradas durante o desenvolvimento de um projeto utilizando a tecnologia descrita.

As funcionalidades mais comuns e também mais importantes serão descritas e observadas utilizando demonstrações de códigos e ilustrações.

3.1. FUNCIONALIDADES DO PRADO NA APLICAÇÃO DESENVOLVIDA

Realizou-se o desenvolvimento de um sistema para controle de aplicações e servidores de acesso, utilizando algumas das principais funcionalidades do PRADO, dentre elas:

- Ferramenta de linha de comando;
- Templates;
- MasterClass;
- Temas;
- TPageService;
- TAuthManager;
- Roles;
- Componentes para entrada de dados;
- Validators;
- DAOs;
- Configurações de aplicação;
- Configurações de páginas;

3.2. ESTRUTURA GERAL

- Sistema Operacional: Windows 7 *Home Premium* (x64);
- Servidor *Web* Apache 2.2.21;
- Servidor de banco de dados MySQL 5.5.16 *Community Edition*;
- Interpretador PHP versão 5.3.8;
- PRADO PHP Framework versão 3.1.10.

3.3. FERRAMENTAS UTILIZADAS

- MySQL Workbench para modelagem do sistema e interação com o banco de dados;
- Visual Paradigm for UML (*Community Edition*);
- AptanaStudio como ambiente de desenvolvimento.

3.4. DESCRIÇÃO DO SISTEMA

A aplicação desenvolvida é um sistema para controle de aplicações para uma empresa qualquer. Existirá um controle de acesso onde administradores poderão gerenciar usuários, aplicações e dados de acesso. Usuários sem a permissão de administrador poderão apenas visualizar os dados de acesso referentes a uma aplicação, dados de acesso estes previamente cadastrados por um administrador.

3.5. REQUISITOS FUNCIONAIS E CASOS DE USO DO SISTEMA

Segue os requisitos funcionais do sistema:

- R1: Relativo ao cadastro, alteração, exclusão e consulta de todos os usuários do sistema;
- R2: Relativo ao cadastro, alteração, exclusão e consulta de todas as aplicações do sistema;
- R3: Relativo ao cadastro, alteração, exclusão e consulta dos servidores de acesso correspondentes a uma aplicação.

Na Figura 25 estão ilustrados os casos de uso do sistema.

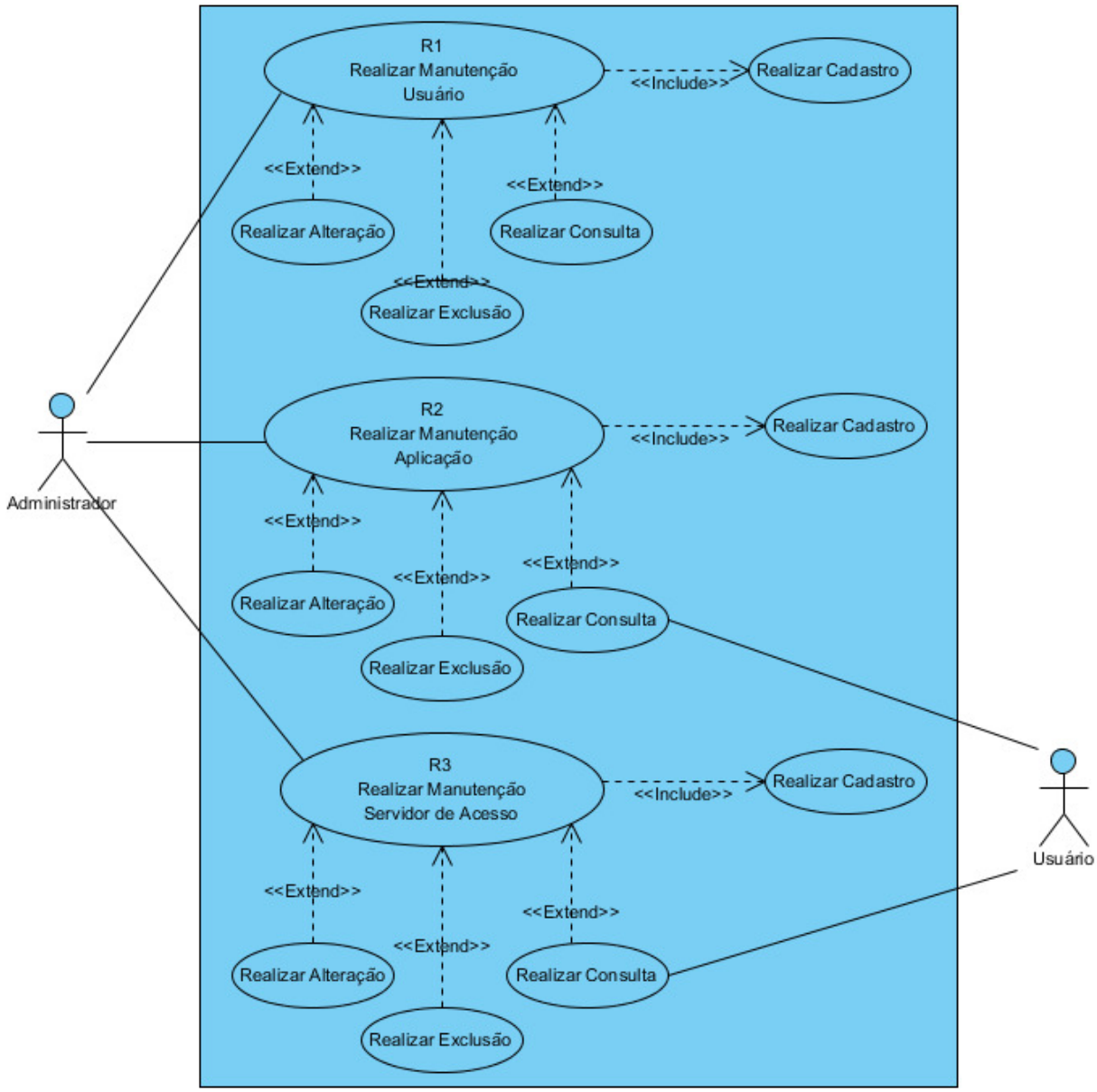

Figura 25 - Casos de Uso do Sistema

3.6. MODELO ENTIDADE RELACIONAMENTO DO SISTEMA

Para um melhor entendimento do sistema desenvolvido, segue na Figura 26 o Modelo Entidade Relacionamento (MER).

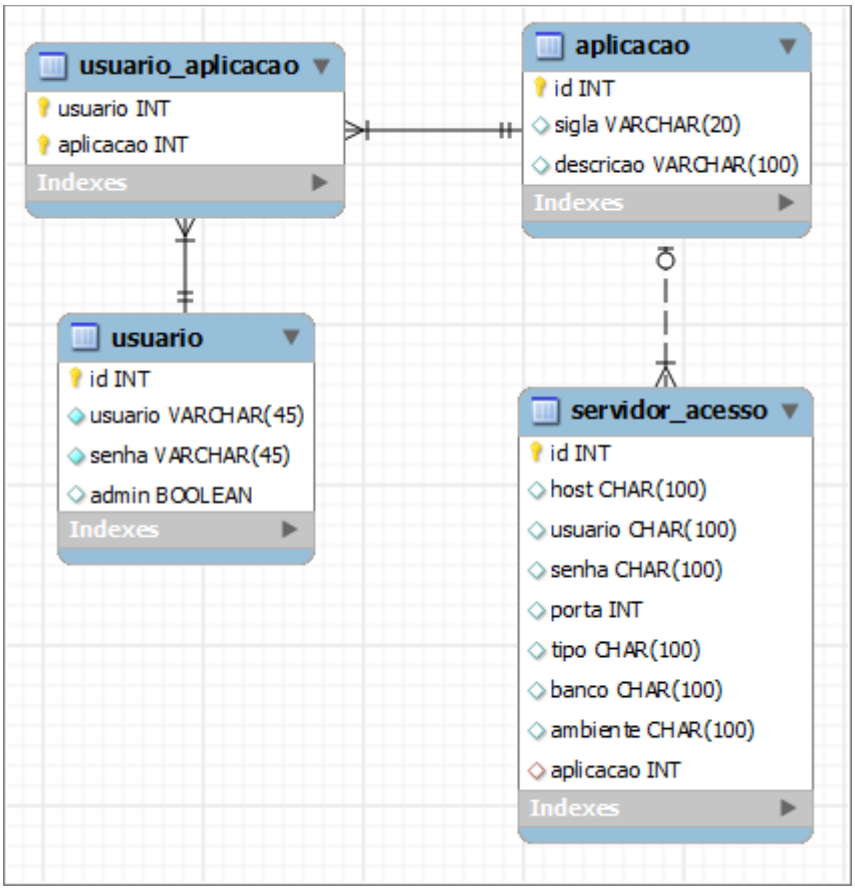

Figura 26 - Modelo Entidade Relacionamento do Sistema desenvolvido

Na tabela `'usuario'` é possível observar o atributo `'admin'` que definirá se o usuário tem permissão de administrador ou não.

4. RESULTADOS E DISCUSSÕES

Neste capítulo está apresentado o desenvolvimento e a apresentação do sistema especificado no capítulo anterior.

4.1. DESENVOLVIMENTO

Considerando as especificações da aplicação descrita anteriormente, neste capítulo serão apresentados além dos principais passos, alguns dos principais elementos do *framework* utilizados e funcionalidades gerais no decorrer do desenvolvimento.

4.1.1. Ferramenta de linha de comando

Como já descrito, esta ferramenta criará o esqueleto com a estrutura de pastas para que o desenvolvedor possa iniciar a programação, sem se preocupar com esta tarefa tediosa.

É importante lembrar que o comando para a criação deste esqueleto deve ser executado na raiz do projeto a ser criado, neste caso, será na pasta `'aplicacoes'`. Esta execução está ilustrada na Figura 27.

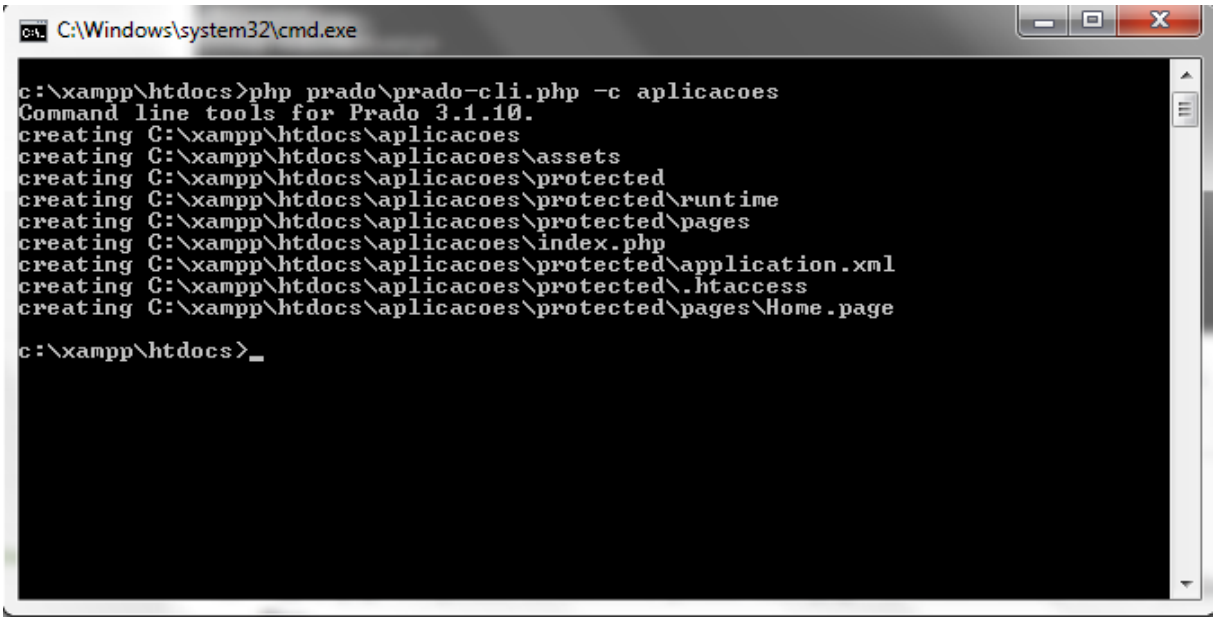

Figura 27 - Criando esqueleto da aplicação

A ferramenta mostra na própria linha de comando quais pastas e arquivos estão sendo criados.

4.1.2. Criando as entidades

O esqueleto da aplicação PRADO está criado e o banco modelado, o próximo passo será criar as entidades que representam esse modelo do banco de dados. Este passo não envolve o *framework* em questão, portanto fica a critério do desenvolvedor implementá-las como bem entender. Na Figura 28 ilustra-se o exemplo de uma destas classes, a classe `Usuario`, onde destacam-se os métodos `getValores()` e `getEditar()`, que são utilizados pelo GenericoDAO.

```php
<?php
class Usuario
{
    private $id;
    private $usuario;
    private $senha;
    private $admin;

    public function getId() {

    public function setId($id) {

    public function getUsuario() {

    public function setUsuario($usuario) {

    public function getSenha() {

    public function setSenha($senha) {

    public function getAdmin() {

    public function setAdmin($admin) {

    public function getTabela() {
        return 'usuario';
    }

    public function getCampos() {
        return ' (usuario, senha, admin)';
    }

    public function getValores() {
        $admin = 'FALSE';
        if($this->admin)
            $admin = 'TRUE';
        return "'" . $this->usuario . "' , " . "md5('"
            . $this->senha . "'), " . $admin;
    }

    public function getEditar() {
        $admin = 'FALSE';
        if($this->admin)
            $admin = 'TRUE';
        return "usuario = '" . $this->getUsuario() .
            "', admin = " . $admin;
    }

    public function arrayToUsuario($array) {
        if(isset($array['id']))
            $this->id = $array['id'];
        if(isset($array['usuario']))
            $this->usuario = $array['usuario'];
        if(isset($array['admin']))
            $this->admin = $array['admin'];
    }
}
```

Figura 28 - Classe Usuario

É importante observar que esta classe possui alguns métodos adicionais, além dos getters e setters dos atributos, que serão utilizados pelos DAOs para construção dos comandos principais dos mesmos.

As demais classes seguem este mesmo modelo.

4.1.3. DAOs

Cada uma das classes da aplicação possui seu respectivo DAO. Estes DAOs serão os responsáveis por fazer a conexão com o banco de dados. Além destes, a aplicação possui um DAO genérico, para funções utilizadas em comum entre os demais. Tarefas como inserção, exclusão, edição e busca serão realizadas por este DAO genérico. Deste modo, os DAOs individuais ficam apenas com tarefas mais específicas da respectiva classe. Para ilustrar, na Figura 29 apresenta-se o DAO da classe `Aplicacao`, seguido pela Figura 30, onde observa-se o `GenericoDAO`.

```php
<?
class AplicacaoDAO extends GenericoDAO {
    private $conexao;
    private $dao;
    private $aplicacao;

    public function AplicacaoDAO() {
        $this->dao = new GenericoDAO();
        $this->aplicacao = new Aplicacao();
    }

    public function salvar($aplicacao) {
        $this->dao->salvar($aplicacao);
    }

    public function editar($aplicacao) {
        $this->dao->editar($aplicacao);
    }

    public function deletar($aplicacao) {
        $this->dao->deletar($aplicacao);
    }

    public function buscaPorId($aplicacao) {
        return $this->dao->buscaPorId($aplicacao);
    }

    public function buscaTodos($objeto=null) {
        return $this->dao->buscaTodos($this->aplicacao);
    }

}
?>
```

Figura 29 – AplicacaoDAO

```php
<?
class GenericoDAO {
    private $conexao;

    public function GenericoDAO() {
        $this->getConexao();
    }
    public function salvar($objeto) {
        $tabela = $objeto->getTabela();
        $campos = $objeto->getCampos();
        $valores = $objeto->getValores();

        $sql = 'INSERT INTO ' . $tabela . $campos . ' VALUES (' . $valores . ');';

        $comando = $this->conexao->createCommand($sql);
        $comando->execute();
    }
    public function editar($objeto) {
        $tabela = $objeto->getTabela();
        $editar = $objeto->getEditar();

        $sql = 'UPDATE ' . $tabela . ' SET ' . $editar . ' WHERE id = ' . $objeto->getId();

        $comando = $this->conexao->createCommand($sql);
        $comando->execute();
    }
    public function deletar($objeto) {
        $sql = 'DELETE FROM ' . $objeto->getTabela() . ' WHERE id = ' . $objeto->getId();

        return $this->conexao->createCommand($sql)->execute();
    }
    public function buscaTodos($objeto) {
        $sql = 'SELECT * FROM ' . $objeto->getTabela();

        return $this->conexao->createCommand($sql)->query()->readAll();
    }
    public function buscaPorId($objeto) {
        $sql = 'SELECT * FROM ' . $objeto->getTabela() . ' WHERE id = ' . $objeto->getId();

        return $this->conexao->createCommand($sql)->query()->read();
    }
    public function getConexao() {
        if($this->conexao == NULL) {
            $this->conexao = new TDbConnection("mysql:host=localhost;dbname=aplicacoes","root","");
            $this->conexao->Active=true;
        }

        return $this->conexao;
    }
}
```

Figura 30 – GenericoDAO

4.1.4. Criando as páginas

Com as classes e seus respectivos DAOs criados, já se tem o necessário para dar início à criação das páginas. Relembrando que todas as classes que se referem a páginas no PRADO descendem da classe `TPage` e possuem um arquivo de *template*, que basicamente montará o leiaute da página.

A primeira página demonstrada será a responsável por permitir inserções de novos usuários na base de dados. No arquivo de *template*, ilustrado na Figura 31, observa-se alguns componentes do PRADO como o `TTextBox` e o `TCheckBox`,

para entrada de dados, e dois validadores: `TRequiredFieldValidor` e `TCustomValidator`. Além destes, há também um componente `TButton` que, ao ser clicado, invocará o método `'salvar'`, definido no evento `'OnClick'` do mesmo. Este arquivo de *template* deve possuir o nome da página seguido pelo sufixo .page. Neste caso, será `NovoUsuario.page`.

```
<html>
    <head><title>Novo Usuário</title></head>
<body>
    <h1>Novo Usuário</h1>
    <fieldset class="login" style="width: 350px;">
        <legend>Preencha todos os campos.</legend>

            <table>
                <tr>
                    <td> <span>Usuário:</span> </td>
                    <Td>
                        <com:TTextBox ID="Usuario" />
                        <com:TRequiredFieldValidator ControlToValidate="Usuario"
                            ErrorMessage="Usuário obrigatório."
                            Display="Dynamic" />
                        <com:TCustomValidator
                            ControlToValidate="Usuario"
                            ErrorMessage="Usuário já existe."
                            Display="Dynamic"
                            OnServerValidate="validaUsuario" />
                    </td>
                </tr>

                <tr>
                    <td> <span>Senha:</span> </td>
                    <Td>
                        <com:TTextBox ID="Senha" TextMode="Password" />
                        <com:TRequiredFieldValidator ControlToValidate="Senha"
                            ErrorMessage="Senha obrigatória."
                            Display="Dynamic" />
                    </td>
                </tr>
                <tr>
                    <td> <span>Admin:</span> </td>
                    <Td> <com:TCheckBox ID="Admin" /> </td>
                </tr>
            </table>

    <com:TButton Text="Salvar" OnClick="salvar" />
    </fieldset>
    <com:TLinkButton Text="Listar Usuários" OnClick="listarUsuarios" CausesValidation="false" SkinID="Navegacao" />

</com:TContent>
</body>
</html>
```

Evento OnClick do botão irá invocar o método 'salvar' no lado do servidor

Figura 31 - Template da página NovoUsuario

A classe responsável por tratar os eventos do *template* deve possuir o mesmo nome do arquivo do mesmo. O nome do arquivo também, este, porém, seguido pelo sufixo .php (no caso será `NovoUsuario.php`). Esta classe responderá pelos eventos acionados pelo usuário no *template*, como o evento `'OnClick'` do componente `TButton`, o qual está ilustrado na Figura 31. A Figura 32 ilustra a classe `'NovoUsuario.php'`.

```php
<?php

class NovoUsuario extends TPage
{
    //Verifica se já existe um usuário com o login fornecido no textbox
    public function validaUsuario($sender, $param) {
        $authManager = $this->Application->getModule('users');
        if($authManager->usernameExists($this->Usuario->Text))
            $param->IsValid = false;
    }

    /**
     * Este evento será invocado quando o usuário clicar no botão 'Salvar'
     * $sender é TButton, componente que invocou este evento
     * $param contém possíveis parâmetros, é uma instância da classe TEventParameter
     */
    public function salvar($sender, $param)
    {
        if ($this->Page->IsValid)  //verifica se passou pelos validadores do template
        {
            //cria novo usuário e seta seus atributos com as informações dos textbox do template
            $usuario = new Usuario();
            $usuario->setUsuario($this->Usuario->Text);
            $usuario->setSenha($this->Senha->Text);
            $usuario->setAdmin($this->Admin->checked);

            $usuarioDAO = new UsuarioDAO();
            $usuarioDAO->salvar($usuario);
            $this->listarUsuarios($sender, $param);
        }
    }

    public function listarUsuarios($sender, $param) {
        $this->Response->redirect('index.php?page=Usuario.ListarUsuarios');
    }
}
?>
```

Figura 32 - Classe da página NovoUsuario

Uma vez cadastrados os usuários, é preciso apresentá-los para que seja possível a visualização destes, bem como a exclusão e edição dos mesmos. O PRADO possui um componente que facilita muito essa apresentação, o qual será utilizado na página `ListarUsuarios`: o `TDataGrid`. Com ele é possível listar os objetos em uma tabela de dados, além de criar comandos facilmente para excluir, editar, entre outros. O *template* da classe `ListarUsuarios` está ilustrado Figura 33.

```
<html>
    <head><title>Listar Usuários</title></head>
<body>
<h1>Usuários</h1>

    <com:TLinkButton Text="Novo Usuário" OnClick="novoUsuario" SkinID="Navegacao" /><br/><br/>

    <com:TDataGrid ID="DgUsuarios" SkinID="DataGrid"
        AutoGenerateColumns="false"
        DataKeyField="id"
        OnItemCreated="itemCreated" OnEditCommand="editItem" OnDeleteCommand="deleteItem">
        <com:TBoundColumn ID="ColunaId" HeaderText="ID" DataField="id"/>
        <com:TBoundColumn ID="ColunaUsuario" HeaderText="Usuário" DataField="usuario" />
        <com:TCheckBoxColumn ID="ColunaAdmin" HeaderText="Admin" DataField="admin" />
        <com:TButtonColumn ID="EditColumn" HeaderText="Editar" Text="Editar" CommandName="edit" />
        <com:TButtonColumn ID="DeleteColumn" HeaderText="Deletar" Text="Deletar" CommandName="delete" />
    </com:TDataGrid>

    <com:TLabel ID="LabelVazio" Text="Não existem usuários cadastrados." Visible="false" SkinID="Vazio" />

</body>
</html>
```

Figura 33 - Template da página ListarUsuarios

É interessante citar algumas propriedades importantes deste componente, como:

- `DataKeyField`: é onde configura-se qual a chave dos objetos listados pelo *datagrid*;
- `OnItemCreated`: evento que será invocado na criação de linha do datagrid;
- `OnEditCommand`: evento que será invocado quando o usuário clicar na coluna de edição;
- `OnDeleteCommand`: evento que será invocado quando o usuário clicar na coluna de exclusão.

Os componentes `TBoundColumn` e `TCheckBoxColumn` são componentes de listagem que são associados aos objetos através da propriedade `'DataField'`, enquanto que o componente `TButtonColumn` irá invocar algum método no lado do servidor, seja de edição ou exclusão.

Estes eventos estão apresentados na Figura 34, que ilustra a classe `ListarUsuarios.php`. Além destes eventos, a classe possui o evento `'onLoad'`, que será invocado durante o carregamento da página e irá carregar os dados apresentados pelo *datagrid*.

```php
<?php

class ListarUsuarios extends TPage
{
    public function onLoad($param)
    {
        parent::onLoad($param);
        if(!$this->IsPostBack)
        {
            $this->carregaUsuarios();
        }
    }

    public function carregaUsuarios() {
        $dataSource = $this->buscaUsuarios();
        $this->DgUsuarios->DataSource = $dataSource;
        $this->DgUsuarios->dataBind();
        if(empty($dataSource)) {
            $this->LabelVazio->Visible = true;
            $this->DgUsuarios->Visible = false;
        } else {
            $this->LabelVazio->Visible = false;
            $this->DgUsuarios->Visible = true;
        }
    }

    public function itemCreated($sender,$param)
    {
        $item=$param->Item;
        if($item->ItemType==='Item' || $item->ItemType==='AlternatingItem' || $item->ItemType==='EditItem') {
            $item->DeleteColumn->Button->Attributes->onclick='if(!confirm(\'Deseja mesmo deletar este usuário?\')) return false;';
        }
    }

    protected function buscaUsuarios() {
        $usuarioDAO = new UsuarioDAO();
        return $usuarioDAO->buscaTodos();
    }

    protected function deletaUsuario($id)
    {
        $usuario = new Usuario();
        $usuario->setId($id);
        $dao = new UsuarioDAO();
        $dao->deletar($usuario);
    }

    public function deleteItem($sender,$param)
    {
        $this->deletaUsuario($this->DgUsuarios->DataKeys[$param->Item->ItemIndex]);
        $this->carregaUsuarios();
    }

    public function editItem($sender,$param)
    {
        $this->Response->redirect('index.php?page=Usuario.EditarUsuario&id=' . $this->DgUsuarios->DataKeys[$param->Item->ItemIndex]);
    }

    public function novoUsuario($sender,$param)
    {
        $this->Response->redirect('index.php?page=Usuario.NovoUsuario');
    }
}

?>
```

Figura 34 - Classe da página ListarUsuarios

Quando o usuário clicar na coluna referente ao botão de edição no *datagrid*, o evento `'editItem'` será invocado no lado do servidor. Este evento foi configurado anteriormente nas propriedades do *datagrid*. Nota-se que, o método manipulador deste evento irá redirecionar o usuário para outra página: a página `EditarUsuario`. Ele irá ainda passar como parâmetro, via URL, a chave do usuário da linha clicada. Chave esta que é obtida através da propriedade `DataKeys`, do *datagrid*.

O arquivo de *template* da página de edição de usuários (`EditarUsuario`) é bastante semelhante ao *template* da página de inserção (`NovoUsuario`), que foi

ilustrada na Figura 31. Como única alteração, o controle referente à senha foi removido, para que o usuário não possa alterar a senha nesta página. Já no arquivo da classe da página, é necessário buscar as informações do usuário e preenchê-las nos campos referentes às mesmas. Para buscar os dados, obtem-se o valor da chave, que vem da URL, e busca-se o objeto na base de dados. Uma vez retornado este objeto, popula-se os controles de entrada com seus valores. A classe `EditarUsuario.php` está ilustrada na Figura 35.

```php
<?php

class EditarUsuario extends TPage
{
    private $_usuario;

    public function onLoad($param)
    {
        parent::onLoad($param);
        if(!$this->IsPostBack) {
            $this->_usuario = new Usuario();
            $this->_usuario->setId($this->Request['id']);
            $usuarioDAO = new UsuarioDAO();
            $array = $usuarioDAO->buscaPorId($this->_usuario);
            $this->_usuario->arrayToUsuario($array);
            $this->Usuario->Text = $this->_usuario->getUsuario();
            $this->Admin->Checked = $this->_usuario->getAdmin();
        }
    }

    public function validaUsuario($sender, $param) {
        $usuarioDAO = new UsuarioDAO();
        $usuario = $usuarioDAO->getPorUsuario($this->Usuario->Text, $this->Request['id']);
        if($usuario != null)
            $param->IsValid = false;
    }

    public function salvar($sender, $param)
    {
        if ($this->Page->IsValid)
        {
            $this->_usuario = new Usuario();
            $this->_usuario->setUsuario($this->Usuario->Text);
            $this->_usuario->setAdmin($this->Admin->checked);
            $this->_usuario->setId($this->Request['id']);

            $usuarioDAO = new UsuarioDAO();
            $usuarioDAO->editar($this->_usuario);
            $this->listarUsuarios($sender, $param);
        }
    }

    public function listarUsuarios($sender, $param) {
        $this->Response->redirect('index.php?page=Usuario.ListarUsuarios');
    }
}
?>
```

Figura 35 - Classe da página EditarUsuario

4.1.5. MasterClass

Agora que se tem um CRUD (*Create, Restore, Update, Delete*) completo, utiliza-se de um conceito do PRADO abstraído do ASP.NET: a MasterClass. Ela será responsável por criar um cabeçalho e um rodapé, que serão utilizados por todas as páginas da aplicação. Para isso, cria-se um arquivo de *template* com o sufixo .tpl e se define nela as informações que serão compartilhadas por todas as páginas. Dentro deste arquivo cria-se o componente `TContentPlaceHolder`, que é onde ficará o conteúdo das demais páginas da aplicação. A MasterClass da aplicação desenvolvida está ilustrada na Figura 36.

```
<html>
<com:THead />
<body>
<com:TForm>
    <div id="page" align="center">

        <div id="header" style="text-align: center;" >
            <com:TLabel SkinID="Header">Sistema de Gerenciamento de Aplicações e Dados de Acesso</com:TLabel>
        </div>
        <div id="menu" padding-top 20px;">
            <com:THyperLink Text="Login" SkinID="Menu"
                NavigateUrl="<%= $this->Service->constructUrl('Login') %>"
                Visible="<%= $this->User->IsGuest %>" />

            <com:THyperLink ID="HlUsuarios" Text="Usuários" SkinID="Menu"
                NavigateUrl="<%= $this->Service->constructUrl('Usuario.ListarUsuarios') %>"
                Visible="<%= $this->User->IsInRole('admin') %>"/>

             <com:THyperLink Text="Aplicações" SkinID="Menu"
                NavigateUrl="<%= $this->Service->constructUrl('Aplicacao.ListarAplicacoes') %>"
                Visible="<%= !$this->User->IsGuest %>" />

            <com:THyperLink Text="Sair" SkinID="Menu"
                NavigateUrl="<%= $this->Service->constructUrl('Logout') %>"
                Visible="<%= !$this->User->IsGuest %>" />
        </div>

        <div id="main" align="center" >
            <com:TContentPlaceHolder ID="Principal" />
        </div>

        <div id="footer" align="center" style="padding-top: 50px;">
            <img src="<%~ utfpr.jpg %>" />
        </div>

    </div>
</com:TForm>
</body>
</html>
```

Figura 36 - Template da MasterClass

É possível ver neste arquivo o menu da aplicação, que estará contido no cabeçalho. Algumas opções estão visíveis para qualquer usuário que não for anônimo, outras apenas para usuários que possuem a role `'admin'`. Estas configurações são feitas na propriedade `'Visible'` dos *links*.

Para fazer uso da MasterClass, é necessário que cada *template* de página que for utilizá-la adicione uma importação para a mesma, como ilustrado na Figura 37.

```
1  <%@ MasterClass="Application.layouts.Principal" Title="Novo Usuário" %>
```

Figura 37 - Importação da MasterClass em um template de página

É necessário também adicionar uma configuração nas configurações da aplicação, o arquivo `application.xml`. Além dessa configuração, na Figura 38 é possível ver a propriedade `'Theme'` configurada. Esta configuração é referente ao tema da aplicação.

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <application id="Aplicacoes" Mode="Debug">
3    <paths>
4      <using namespace="Application.daos.*" />
5      <using namespace="Application.entidades.*" />
6      <using namespace="Application.modulos.*" />
7      <using namespace="System.Data.*" />
8      <using namespace="System.Data.ActiveRecord.*" />
9      <using namespace="System.Security.*" />
10     <using namespace="System.Web.UI.ActiveControls.*" />
11   </paths>
12   <services>
13     <service id="page" class="TPageService" DefaultPage="Home">
14       <pages MasterClass="Application.layouts.Principal" Theme="Componentes" />
15     </service>
16   </services>
17 </application>
```

Figura 38 - Configurações da aplicação

4.1.6. Tema

Uma vez configurado nas configurações de aplicação, para criar um tema basta criar um arquivo com o sufixo `.skin` com o mesmo nome configurado na propriedade `'Theme'` anteriormente. Este arquivo conterá configurações de estilo personalizadas para os componentes desejados, e, para utilizá-los em cada componente é necessário adicionar a propriedade `SkinID`, configurando com o mesmo valor do arquivo de temas. O arquivo de temas da aplicação desenvolvida está ilustrado na Figura 39.

```
<com:TLabel SkinID="Header"
    BackColor="black"
    ForeColor="#F5C500"
    BorderColor="#F5C500"
    BorderStyle = "solid"
    BorderWidth = "1"
    Font.Size="24"
    Style="padding-bottom: 5px; padding-top: 5px; padding-right: 5px; padding-left: 5px;"
    Width="16"
    />
<com:THyperLink SkinID="Menu"
    BackColor="#F5C500"
    ForeColor="black"
    Style="text-decoration:none;"
    />
<com:TLinkButton SkinID="Navegacao"
    ForeColor="Red"
    Style="text-decoration:underline;"
    />
<com:TLinkButton SkinID="Menu"
    BackColor="#F5C500"
    ForeColor="black"
    Style="text-decoration:none;"
    />
<com:TLabel SkinID="Vazio"
    ForeColor="red"
    Style="text-decoration:none;"
    />
<com:TDataGrid SkinID="DataGrid" CellPadding="2"
    HeaderStyle.BackColor="black"
    HeaderStyle.ForeColor="white"
    ItemStyle.BackColor="#EDD35C"
    ItemStyle.Font.Italic="true"
    AlternatingItemStyle.BackColor="#F5C500" />
```

Figura 39 - Arquivo de temas

4.1.7. Configurações das páginas

O arquivo de configurações das páginas foi alterado para utilizar módulo de controle de usuários personalizado, além de serem definidas as *roles* para dizer quem tem acesso ao quê. A Figura 40 mostra esse arquivo de configurações.

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <modules>
    <module id="users" class="AplicacaoUserMAnager" />
    <module id="auth" class="TAuthManager" UserManager="users" LoginPage="Login" />
  </modules>
  <authorization>
    <allow pages="Login" users="?" />
    <allow pages="Home" users="?" />
    <deny pages="ListarUsuarios, NovoUsuario, EditarUsuario, NovaAplicacao, EditarAplicacao,
                 NovoServidorAcesso, EditarServidorAcesso" roles="normal" />
    <allow roles="normal" />
    <allow roles="admin" />
    <deny users="*" />
  </authorization>
</configuration>
```

Figura 40 - Configurações das páginas

Lembrando que este arquivo foi criado automaticamente através da ferramenta de linha de comandos, se chama `config.xml` e se encontra dentro do diretório `'pages'` da aplicação.

4.1.8. Controle de usuários

O PRADO fornece a classe `TUserManager` para controle de usuários, mas é possível implementar sua própria classe. Na aplicação desenvolvida foi programada a classe `'AplicacaoUserManager'`. Para que se possa utilizá-la para controle de usuários de uma aplicação PRADO é preciso que ela descenda da classe `TModule`, o que a identificará como um módulo do PRADO e que implemente a interface `IUserManager`, que contém métodos para este controle de usuários, como o método de autenticação `'login'`. Personaliza-se estes métodos para que acessem o DAO da classe `Usuario`, obtendo assim informações da base de dados. Esta classe está ilustrada na Figura 41.

```php
<?
class AplicacaoUserManager extends TModule implements IUserManager
{
    public function getGuestName() {
        return 'Anônimo';
    }
    public function getUser($login=null) {
        $usuario = new TUser($this);
        $usuario->setIsGuest(true);
        if($login !== null && $this->usernameExists($login))
        {
            $usuario->setIsGuest(false);
            $usuario->setName($login);
            $usuarioDAO = new UsuarioDAO();
            $usuarioLogado = $usuarioDAO->getPorUsuario($login);
            if($usuarioLogado->getAdmin())
                $usuario->setRoles(array('admin'));
            else
                $usuario->setRoles(array('normal'));
        }
        return $usuario;
    }
    public function addNewUser($login, $senha, $admin) {
        $usuario = new Usuario();
        $usuario->usuario = $login;
        $usuario->senha = $senha;
        $usuario->admin = $admin;
        $usuarioDAO = new UsuarioDAO();
        $usuarioDAO->salvar($usuario);
    }
    public function usernameExists($login) {
        $usuarioDAO = new UsuarioDAO();
        $usuario = $usuarioDAO->getPorUsuario($login);

        if($usuario != null)
            return true;
        return false;
    }
    public function validateUser($login, $senha) {
        $usuarioDAO = new UsuarioDAO();
        return $usuarioDAO->autentica($login, $senha);
    }
    public function getIsAdmin() {
        return $this->User->IsInRole('admin');
    }
    public function getUserFromCookie($cookie) {}

    public function saveUserToCookie($cookie) {}
}
?>
```

Figura 41 - Classe AplicacaoUserManager

4.1.9. Login

Uma vez configurado o módulo `'auth'` nas configurações de páginas, efetuar login no sistema se torna uma tarefa bastante simples. Para isso cria-se uma página chamada `'Login'` e, no *template* da mesma, cria-se dois componentes para entrada de dados referentes ao usuário e senha. Além dos validadores de campos obrigatórios foi adicionado também um validador personalizado, que invocará um método do lado do servidor, no nosso caso será o método `'validaLogin'`. O arquivo de *template* desta página está ilustrado na Figura 42.

```
1  <%@ MasterClass="Application.layouts.Principal" Title="Novo Usuário" %>
2
3  <com:TContent ID="Principal">
4      <h1 class="login">Efetue login para ter acesso ao sistema</h1>
5      <fieldset style="width: 350px;">
6          <legend>Entre com seu usuário e senha:</legend>
7          <table>
8              <tr>
9                  <td> <com:TLabel ForControl="Usuario" Text="Usuário:" /> </td>
10                 <td>
11                     <com:TTextBox ID="Usuario" MaxLength="40" />
12
13                     <com:TRequiredFieldValidator
14                         ControlToValidate="Usuario"
15                         Display="Dynamic"
16                         ErrorMessage="Usuário obrigatório." />
17
18                     <com:TCustomValidator
19                         ControlToValidate="Usuario"
20                         Display="Dynamic"
21                         ErrorMessage="Usuário e/ou senha inválido(s)."
22                         OnServerValidate="validaLogin" />
23                 </td>
24             </tr>
25             <tr>
26                 <Td> <com:TLabel ForControl="Senha" Text="Senha:" /> </td>
27                 <td>
28                     <com:TTextBox ID="Senha" TextMode="Password" MaxLength="20" />
29
30                     <com:TRequiredFieldValidator
31                         ControlToValidate="Senha"
32                         Display="Dynamic"
33                         ErrorMessage="Senha obrigatório." />
34                 </td>
35             </tr>
36         </table>
37         <div class="login-button">
38             <com:TButton Text="Login" OnClick="efetuarLogin" />
39         </div>
40         </fieldset>
41
42 </com:TContent>
```

Validação será efetuada no lado do servidor

Evento invocado ao clicar no botão de login

Figura 42 - Template da página de Login

Ao clicar no botão de login, antes de invocar o método `'efetuarLogin'` definido no evento `'OnClick'` do botão, a página irá invocar o método `'validaLogin'`, configurado no validador personalizado. Caso obtenha sucesso, o método `'efetuarLogin'` será invocado, caso contrário a mensagem definida no validador será exibida.

Na Figura 43 está a ilustração da classe desta página `'Login'`, onde só existem dois métodos: os métodos definidos no *template*.

```
1  <?php
2  class Login extends TPage
3  {
4      public function validaLogin($sender,$param) {
5          $authManager=$this->Application->getModule('auth');
6          if(!$authManager->login($this->Usuario->Text, $this->Senha->Text))
7              $param->IsValid=false;
8      }
9
10     public function efetuarLogin($sender,$param)
11     {
12         if($this->Page->IsValid)
13             $this->Response->redirect('index.php?page=Home');
14     }
15 }
16 ?>
```

Figura 43 - Classe da página Login

Esta classe irá utilizar o módulo `'auth'` definido anteriormente nas configurações de páginas. Este módulo se irá delegar o método de login para a classe personalizada de controle de usuários. Ela é quem irá manter o usuário autenticado na sessão e este, uma vez autenticado, estará disponível em qualquer momento da aplicação a partir do método `$this->User`.

4.2. EXECUTANDO A APLICAÇÃO

Ao entrar na aplicação desenvolvida o usuário será redirecionado para a página `'Home'`, a página inicial do sistema. Já nesta página é possível observar que a única opção disponível no menu é a opção `'Login'`, visto que ao entrar na aplicação o usuário é considerado como anônimo. Pode-se observar na Figura 44 o conceito da MasterClass em ação.

Figura 44 - Página inicial

Ao entrar na página `Login` e efetuar um clique no botão `'Login'` sem preencher as informações, os validadores entrarão em ação e apresentarão as mensagens de erro, dizendo que os campos são obrigatórios, o mesmo aconteceria ao tentar entrar no sistema com usuários não existentes ou senhas que não conferem. A Figura 45 ilustra a tentativa de autenticação sem dados.

Figura 45 - Validadores em ação

Após efetuar a autenticação com um usuário com permissão de administrador todas as opções do menu estarão disponíveis, como ilustrado na Figura 46.

Figura 46 - Autenticado como administrador

Enquanto que, ao se autenticar com usuário normal, apenas as opções `'Aplicações'` e `'Sair'` estarão disponíveis. Estas configurações de visualização do menu foram feitas diretamente no arquivo de *template* da MasterClass, e foram ilustradas na Figura 33.

Ao clicar no menu `'Usuários'` a aplicação irá redirecionar para a página de listagem de usuários: a página `ListarUsuarios`. Ela irá listar todos os usuários cadastrados no sistema, com as opções para exclusão e edição, além de um *link* para que seja possível criar um novo usuário. É importante lembrar que esta opção só está disponível para usuários com permissão de administrador e, caso um usuário sem esta permissão tente acessá-la diretamente pela URL, o mesmo será redirecionado para a página inicial. Na Figura 47 observa-se a ilustração desta página de listagem de usuários.

Figura 47 - Listagem de Usuários

Por fim, a página para inserção de um novo usuário está ilustrada na Figura 48.

Figura 48 - Página para criação de novos usuários

5. CONSIDERAÇÕES FINAIS

Esse capítulo tratará das conclusões finais que foram abstraídas durante o desenvolvimento do projeto. Também serão sugeridas algumas idéias para pessoas que leram o trabalho e desejam dar continuidade no mesmo.

5.1. CONCLUSÃO

Através da aplicação desenvolvida ficam incontestáveis as facilidades e agilidades que o *framework PRADO PHP Framework* pode fornecer. Ele é capaz de prover funcionalidades que ajudam o desenvolvedor desde o início até o fim de cada projeto.

Os componentes que o *framework* fornece podem poupar um trabalho enorme fornecendo ferramentas já prontas que são flexíveis o suficiente para se adaptar em qualquer tipo de aplicação. Perde-se pouco tempo para aprendê-los. Não só para os componentes, mas para o *framework* como um todo, mas o resultado é compensador. Os componentes de entrada de dados e seus validadores tornam as tarefas repetitivas em um sistema algo menos tedioso e muito mais fácil e rápido de ser desenvolvido.

Além destes componentes, os conceitos utilizados pelo *framework* para autorização podem agilizar muito o processo que diz quem pode acessar o quê. Uma simples adição de uma regra nas configurações das páginas pode bloquear ou dar acesso a todas as páginas de um sistema para um usuário específico, ou um grupo inteiro de usuários.

O *PRADO* fornece ainda facilidades quando feito acessos ao banco de dados. De uma maneira fácil é possível fazer operações sem retorno como inserções ou remoções e também operações com retorno, como em um *select*.

O *framework* também não deixa a desejar na customização do leiaute de um sistema. Utilizando o conceito de *MasterClass* é possível tornar a manutenção deste leiaute muito mais simples. Alterando esta única classe pode-se alterar todas as páginas do sistema. Além deste, com a customização de temas podemos estilizar todos os componentes que o *framework* provê para o desenvolvedor, de uma maneira fácil e intuitiva.

É importante citar que o *framework* possui uma documentação bastante poderosa, que vai desde exemplos à documentação individual de cada classe que ele fornece para o desenvolvedor.

5.2. TRABALHOS FUTUROS

Como sugestão para trabalhos futuros, fica a ideia de explorar melhor os recursos de personalização de módulos e *services* do PRADO PHP Framework, além de explorar os controles ativos (AJAX) do mesmo.

REFERÊNCIAS BIBLIOGRÁFICAS

DALL'OGLIO**. PHP: Programando com Orientação a Objetos.** São Paulo: Novatec, 2007.

MELO, Alexandre Altair, NASCIMENTO, Mauricio G.F. **PHP Profissional**. São Paulo: Novatec, 2007.

MINETTO, Elton L. **Frameworks para desenvolvimento em PHP**. São Paulo: Novatec, 2007.

PHP. **A História do PHP.** Disponível em: <http://www.php.net/manual/pt_BR/history.php.php>. Acesso em: 08 ago. 2011.

PHP. **O que é PHP?.** Disponível em: < http://www.php.net/manual/pt_BR/intro-whatis.php >. Acesso em: 10 jul. 2011.

PORTER, Michael E. **Vantagem Competitiva**. Rio de Janeiro: Campus, 1989.

PRADO PHP Framework. **About PRADO.** Disponível em: <http://www.pradosoft.com/about/>. Acesso em 10 ago. 2011.

PRADO PHP Framework. **PRADO QuickStart Tutorial.** Disponível em: <http://www.pradosoft.com/demos/quickstart/>. Acesso em 05 set. 2011.

PRADO PHP Framework. **What is PRADO?.** Disponível em: < http://www.pradosoft.com/demos/quickstart/?page=GettingStarted.AboutPrado >. Acesso em 05 set. 2011.

QIANG, Xue. Disponível em: < http://www.pradosoft.com/demos/quickstart/?page=GettingStarted.AboutPrado >. Acesso em: 17 set. 2011.

SOARES, Bruno A. L.. **Aprendendo a linguagem PHP.**São Paulo: Ciência Moderna, 2007.

THOMSON, Laura, Welling, Luke. **PHP e MySQL: Desenvolvimento Web**. Rio de Janeiro: Campus, 2005.

TIOBE Software. **Tiobe Index.** Disponível em: <http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html>. Acesso em 10 jul. 2011.

VASWANI, Vikram. **PHP Programming Solutions.** McGraw-Hill Osborne Media, 2007.