



ADIANTI FRAMEWORK PARA PHP

10^a edição

Licenciado para MARCOS ANTONIO RAFAEL DA FONSECA -



PABLO DALL'OGLIO

Copyright © 2012 de Pablo Dall'Oglio.

Todos os direitos reservados e protegidos pela Lei 9.160 de 19/02/1998.

É proibida a reprodução desta obra, mesmo que parcial, por qualquer processo, sem prévia autorização por escrito, do autor.

Dezembro 2012 - 1a. edição

Maio 2013- 2a. edição

Setembro 2013- 3a. edição

Maio 2014- 4a. edição

Janeiro 2015- 5a. edição

Setembro 2015- 6a. edição

Janeiro 2016- 7a. edição

Janeiro 2017- 8a. edição

Agosto 2018 - 9a edição

Novembro 2019 - 10a edição

Editor: Pablo Dall'Oglio

Revisão gramatical: Fernanda Dall'Oglio

Capa: Pablo Dall'Oglio

Adianti Soluções Ltda.

www.adianti.com.br

Dados internacionais de Catalogação na Publicação (CIP)

(Câmara Brasileira do Livro, SP, Brasil)

Dall'Oglio, Pablo
Adianti Framework para PHP / Pablo
Dall'Oglio. Lajeado: Edição do autor, 2019.

Bibliografia.
ISBN 978-85-914354-9-4

1. Linguagens de programação 2. Frameworks 3.
PHP 4. Adianti Framework

Sumário

1	Introdução.....	12
1.1	Características.....	12
1.2	Arquitetura.....	16
1.3	Estrutura de diretórios.....	17
1.4	Projetos Adianti.....	18
1.5	Diferenciais do Adianti Framework.....	19
1.6	Começar pelo Framework ou pelo Template?.....	20
1.7	Ganhar produtividade com o Studio ou Builder?.....	21
1.8	Um controlador de página.....	22
1.9	A página de entrada.....	22
1.9.1	O index.....	23
1.9.2	O init.....	25
1.9.3	O layout.....	26
1.9.4	O menu.....	27
1.9.5	As bibliotecas.....	28
1.9.6	O engine.....	29
1.10	Adianti Tutor.....	30
2	Instalação e configuração.....	31
2.1	Instalação do ambiente.....	31
2.1.1	Ambiente Linux.....	31
2.1.2	Ambiente Windows.....	32
2.1.3	Ambiente MacOS.....	33
2.2	Instalação do framework.....	34
2.2.1	Ambiente Linux.....	34
2.2.2	Ambiente Windows.....	34
2.2.3	Ambiente MacOS.....	34
3	Modelos e persistência.....	35
3.1	Modelo utilizado.....	35
3.1.1	Modelo de classes.....	35
3.1.2	Modelo relacional.....	37
3.2	Configuração e acesso ao banco de dados.....	38
3.2.1	Criação do banco de dados.....	38
3.2.2	Configuração do acesso.....	39
3.2.3	Como executar os exemplos.....	41
3.2.4	Transações e queries manuais.....	42
3.2.5	Queries manuais com prepared statements.....	43
3.2.6	Conexão manual.....	44
3.3	Manipulação de objetos.....	45
3.3.1	O padrão Active Record.....	45
3.3.2	Definindo uma classe Active Record.....	46
3.3.3	Gravação de novo objeto.....	47
3.3.4	Carregamento de objeto.....	48
3.3.5	O padrão Lazy Load.....	50
3.3.6	Alteração de objeto.....	52
3.3.7	Registro de log.....	53
3.3.8	Encapsulamento.....	55
3.3.9	Exclusão de objeto.....	57

3.3.10 Primeiros e últimos Identificadores.....	57
3.3.11 Conversão entre Active Record e Array.....	58
3.3.12 Conversão para JSON.....	59
3.3.13 Hook methods.....	60
3.3.14 Renderização e cálculos.....	62
3.3.15 CreatedAt e UpdatedAt.....	62
3.4 Manipulação de coleções.....	63
3.4.1 O padrão Repository.....	63
3.4.2 API de critérios.....	64
3.4.3 Contagem de coleções.....	65
3.4.4 Carregamento de coleções.....	67
3.4.5 Carregamento paginado e ordenado.....	69
3.4.6 Carregamento de listas simples de dados.....	70
3.4.7 Agregações.....	71
3.4.8 Alteração de coleções.....	72
3.4.9 Exclusão de coleções.....	75
3.4.10 Transformação de coleções.....	76
3.5 Relacionamentos entre objetos.....	77
3.5.1 Associação.....	77
3.5.2 Composição.....	80
3.5.3 Agregação.....	85
3.5.4 Atalhos em relacionamentos.....	90
4 Componentes de apresentação.....	92
4.1 Conceitos básicos.....	92
4.1.1 Controlador de páginas.....	92
4.1.2 Ações.....	95
4.2 Páginas.....	97
4.2.1 Página com HTML.....	97
4.2.2 Página com PDF embutido.....	98
4.2.3 Página com conteúdo externo.....	99
4.2.4 Página na forma de Janela.....	100
4.2.5 Abrindo janela sob demanda.....	101
4.2.6 Janela Modal.....	101
4.2.7 Cortina lateral.....	103
4.3 Containers.....	105
4.3.1 Tabela.....	105
4.3.2 Lidando com colunas em tabelas.....	107
4.3.3 Trabalhando com células mescladas.....	108
4.3.4 Painel.....	110
4.3.5 Notebook.....	111
4.3.6 Notebook Bootstrap.....	112
4.3.7 Panel group.....	113
4.3.8 Scroll.....	114
4.3.9 Frame.....	115
4.3.10 Caixas horizontais e verticais.....	117
4.4 Diálogos.....	118
4.4.1 Informação.....	118
4.4.2 Atenção.....	119
4.4.3 Erro.....	120
4.4.4 Questionamento.....	120
4.4.5 Toast.....	122
4.4.6 Input.....	123
4.5 Formulários.....	124
4.5.1 Componentes para formulários.....	124
4.5.2 Formulário manual.....	126
4.5.3 Formulários Bootstrap.....	129
4.5.4 Formulário Bootstrap em colunas.....	132
4.5.5 Formulário Bootstrap com labels acima.....	134

4.5.6 Postagem estática de formulários.....	137
4.5.7 Formulário com lista de campos.....	139
4.5.8 Formulário com check list.....	142
4.5.9 Estilos de botão.....	145
4.5.10 Máscaras de input.....	146
4.5.11 Validações.....	148
4.5.12 Criando um validador.....	150
4.5.13 Seleções estáticas.....	152
4.5.14 Seleções manuais.....	155
4.5.15 Seleções automáticas.....	157
4.5.16 Interações dinâmicas.....	160
4.5.17 Habilitando e desabilitando campos.....	164
4.5.18 Botão de busca de registros.....	166
4.5.19 Edição de HTML.....	169
4.5.20 Listas de ordenação.....	170
4.5.21 Formulários MVC reutilizáveis.....	173
4.6 Datagrids.....	175
4.6.1 Datagrids.....	176
4.6.2 Datagrids Bootstrap.....	179
4.6.3 Datagrids com grupos de ações.....	181
4.6.4 Datagrids com ações condicionais.....	182
4.6.5 Datagrids com Popover.....	184
4.6.6 Datagrids com rolagem horizontal.....	186
4.6.7 Datagrids com rolagem vertical.....	187
4.6.8 Datagrid com colunas escondidas.....	189
4.6.9 Datagrid com datatables.....	191
4.6.10 Datagrid com busca rápida.....	192
4.6.11 Datagrid com formatação de colunas.....	194
4.6.12 Datagrids com links.....	196
4.6.13 Datagrids com cálculos.....	198
4.6.14 Datagrids com imagem.....	200
4.6.15 Datagrids com barra de progresso.....	202
4.6.16 Exportação de datagrids.....	204
4.6.17 Datagrids com campos de entrada.....	206
4.6.18 Datagrids com checkbox.....	209
4.7 Transições de páginas.....	212
4.7.1 Passo a passo entre formulários diferentes.....	212
4.7.2 Wizard passo a passo.....	216
4.8 Utilitários.....	223
4.8.1 Passo a passo.....	223
4.8.2 Visão de ícones.....	224
4.8.3 Linha do tempo.....	226
4.8.4 Visão de cards.....	228
4.8.5 Kanban.....	230
4.8.6 Árvore.....	233
4.8.7 Calendário.....	234
4.9 Templates e novos componentes.....	237
4.9.1 Template View básico.....	237
4.9.2 Template View avançado.....	240
4.9.3 Template View com matrizes.....	245
4.9.4 Criando componentes.....	248
4.10 Relatórios.....	251
4.10.1 Relatórios tabulares.....	251
4.10.2 Relatório sobre consulta SQL.....	256
4.10.3 Relatório sobre View.....	260
4.10.4 Conversão de Templates para PDF.....	261
4.10.5 Fatura em tela e para impressão.....	263
4.11 Gráficos.....	267
4.11.1 Gráfico de linhas.....	267

4.11.2	Gráfico de barras.....	269
4.11.3	Gráfico de pizza.....	270
4.11.4	Dashboard.....	271
4.12	Etiquetas.....	273
4.12.1	Etiquetas de Códigos de barras.....	273
4.12.2	Etiquetas de QRCode.....	276
4.12.3	Etiquetas em tela.....	279
5	Organização e controle.....	281
5.1	Cadastros padronizados.....	281
5.1.1	Formulário padronizado.....	281
5.1.2	Datagrid padronizada.....	283
5.1.3	Formulário comdatagrid padronizada.....	287
5.2	Cadastros manuais.....	290
5.2.1	Formulário manual.....	290
5.2.2	Datagrid manual.....	293
5.2.3	Formulário comdatagrid manual.....	299
5.3	Cadastros completos.....	305
5.3.1	Formulário de clientes.....	305
5.3.2	Listagem de clientes.....	313
5.3.3	Formulário de Produtos.....	318
5.3.4	Listagem de Produtos.....	322
5.3.5	Formulário mestre-detalhe de vendas.....	325
5.3.6	Listagem de vendas.....	334
5.4	Telas de consulta.....	337
5.4.1	Consulta o status de um cliente...	337
5.5	Operações em lote.....	342
5.5.1	Edição de registros em lote.....	342
5.5.2	Exclusão de registros em lote.....	346
5.5.3	Seleção de registros em lote.....	351
6	Template para criação de sistemas.....	356
6.1	Visão geral.....	356
6.1.1	Formulário de Login.....	358
6.1.2	Módulos.....	359
6.1.3	Estrutura de diretórios.....	361
6.1.4	O application.ini.....	362
6.1.5	Menu da aplicação.....	363
6.1.6	Layout e temas.....	364
6.2	Módulo Administração.....	367
6.2.1	Diagrama de classes.....	367
6.2.2	Modelo relacional.....	368
6.2.3	Formulário de login.....	370
6.2.4	Multi unidade.....	373
6.2.5	Multi database.....	375
6.2.6	Internacionalização.....	376
6.2.7	Multi idioma.....	378
6.2.8	Definindo uma autenticação alternativa (LDAP)...	379
6.2.9	O index.....	380
6.2.10	O engine.....	382
6.2.11	Visão pública.....	384
6.2.12	Cadastro de programas.....	385
6.2.13	Cadastro de grupos.....	386
6.2.14	Cadastro de usuários.....	387
6.2.15	Cadastro de unidades.....	388
6.2.16	Database explorer.....	389
6.2.17	Painel de SQL.....	389
6.2.18	Preferências.....	390
6.2.19	Editor de menu.....	391
6.3	Configuração e depuração.....	392

6.3.1	PHP Modules.....	392
6.3.2	PHP Info.....	392
6.3.3	Debug console.....	393
6.4	Módulo Logs.....	394
6.4.1	Classes de Modelo.....	394
6.4.2	Classes de Serviço.....	394
6.4.3	Modelo relacional.....	395
6.4.4	Logs de acesso.....	396
6.4.5	Logs de SQL.....	397
6.4.6	Logs de alterações.....	398
6.4.7	Logs de requisição.....	400
6.5	Módulo Comunicação.....	401
6.5.1	Diagrama de classes.....	401
6.5.2	Modelo relacional.....	402
6.5.3	Gestão de documentos.....	403
6.5.4	Troca de mensagens.....	405
6.5.5	Notificações do sistema.....	405
6.6	Dicas de utilização.....	407
6.6.1	Novos temas.....	407
6.6.2	Alterando as cores do tema 3.....	408
6.6.3	Alterando as cores do tema 4.....	409
6.6.4	Criando um programa dentro do Template.....	410
6.6.5	Práticas responsivas.....	411
7	Estudos de caso.....	415
7.1	Aplicação Library.....	415
7.1.1	Conteúdo da aplicação.....	416
7.1.2	Diagrama de classes.....	416
7.1.3	Modelo relacional.....	417
7.1.4	Diagrama de casos de uso.....	418
7.1.5	Especificação dos casos de uso.....	419
7.1.6	Login e perfis.....	419
7.2	Aplicação Changeman.....	420
7.2.1	Conteúdo da aplicação.....	421
7.2.2	Diagrama de classes.....	421
7.2.3	Modelo relacional.....	422
7.2.4	Diagrama de casos de uso.....	423
7.2.5	Especificação dos casos de uso.....	424
7.2.6	Login e perfis.....	425
8	Integrações.....	426
8.1	Rotas amigáveis.....	426
8.2	Serviços REST.....	429
8.3	Serviços RESTful.....	436
8.4	Serviços RESTful seguro com token JWT.....	447
8.5	Manipulando usuários por REST.....	451
8.6	Envio de emails.....	454
8.7	Pacotes Composer.....	455
8.8	PWA Manifest.....	457

Dedicatória

Dedico esta obra a todos aqueles que acreditam no meu trabalho e me motivam a continuar criando soluções e escrevendo livros. Obrigado do fundo do coração. Eu respondo e guardo cada e-mail de vocês com muita felicidade.

Agradecimentos

Agradeço primeiramente a Deus, que tem me dado força todos os dias para cumprir minha missão pessoal, que eu acredito ser construir e ensinar. Nesse sentido, tenho me empenhado em construir ferramentas que possam não apenas auxiliar em meus projetos, mas que também possam auxiliar pessoas como você. Em segundo lugar, agradeço a minha esposa por compreender minha atuação profissional e minhas madrugadas de programação; por me amar incondicionalmente e ter gerado nossa pequena amada Ana Júlia, que trouxe tanta vida para nosso lar. Também agradeço aos meus pais e meus avós pela educação e por apostarem em mim, e minha irmã por estar sempre ao meu lado. E aos meus amigos, que não são muitos, mas são verdadeiros irmãos pois sei que sempre posso contar com eles.

Prefácio

Seja bem-vindo ao mundo do Adianti Framework. Primeiro, deixe-me apresentar: Me chamo Pablo Dall'Oglio e desenvolvo sistemas desde 1994. Mas o que importa é que trabalho com PHP desde 2000. Antes desse período trabalhei com Clipper de 1994 a 1998 e com Delphi de 1998 a 2000. Quando conheci o PHP em 2000, não eram muitas as pessoas ligadas à tecnologia que conheciam o PHP aqui no Brasil. Fomos precursores no seu uso devido ao Cesar Brod, na época responsável pela TI da Univates (universidade na qual eu trabalhava). Ele apostou no PHP para construirmos um completo ERP acadêmico, e assim nascia o SAGU (Sistema Aberto de Gestão Unificada). Naquela época programávamos rapidamente, mas sem nenhum padrão. Os códigos misturavam HTML, SQL e era um *copy and paste* desenfreado. Bom, o sistema fez até um relativo sucesso como software livre mas sua manutenção era horrível. Depois de um tempo eu me desvinculei do projeto e comecei a trabalhar com outras coisas. Dentre estas me interessava muito automatizar a parte de geração de relatórios. Foi então que, inspirado no Crystal Reports, eu criei o Agata Report, um gerador de relatórios em PHP.

Quando era aluno universitário lá pelo ano de 2004, conheci Design Patterns na disciplina de Engenharia de Software na Unisinos com o professor Sérgio Crespo e desde lá assuntos como padrões de projeto e arquiteturas de software fazem parte do meu dia a dia. Hoje estou do outro lado, ensinando estes assuntos para os alunos nas disciplinas de Engenharia de Software. Bom, mas lá pelo ano de 2006, quando eu estava para me formar, precisava escrever uma aplicação para o TCC: um sistema para gerenciar projetos em Extreme Programming. Naquele momento não tínhamos essa enormidade de frameworks como temos hoje. Como estava com os padrões de projeto fresquinhos na mente e com uma vontade grande de automatizar tudo que fosse possível (ou uma preguiça de escrever muito código), resolvi implementar um framework MVC em PHP. O projeto deu muito certo para o TCC e acabei mostrando para outros profissionais à minha volta.

Na mesma época eu era consultor de projetos na Univates e o pessoal de lá gostou muito do framework. Como o SAGU já não estava aguentando as demandas, a Univates precisava escrever um novo ERP acadêmico e o pessoal resolveu adotar o meu framework, na época e ainda hoje chamado simplesmente de Core. O novo sistema

ERP (chamado de Alfa) foi um sucesso e desde lá mais de 20 diferentes projetos foram desenvolvidos internamente na Univates usando o Core. O Core era e ainda é muito bom, mas algumas ideias minhas foram amadurecendo com o tempo e o framework tinha algumas coisas que eu não gostava de como havia implementado. Então, lá pelo ano de 2008 eu joguei o código fora e comecei a escrever um novo framework, desta vez usando o nome de Adianti Framework para dar maior ênfase à empresa que tenho constituída desde 2006, a Adianti Solutions. Bom, em 2008 comecei o mestrado e logo que conclui em 2010 comecei a lecionar no ensino superior, sendo que somente no ano de 2012 consegui lançar a primeira versão pública do framework, bem como o primeiro livro. E isso responde em parte a pergunta: e por que mais um framework? Na verdade o Adianti Framework representa a segunda geração de um framework que já vem sendo desenvolvido desde 2006, quando a grande maioria dos frameworks existentes na atualidade ainda nem existiam. Desde o seu lançamento ele vem se consolidando como um framework especializado no desenvolvimento de sistemas de gestão (ERP), com diversas facilidades para tal. O retorno da comunidade e das empresas que o utilizam tem sido fantástico desde seu lançamento como projeto em Software Livre, e a comunidade cresce a cada dia.

As principais características do Adianti Framework são: ser um framework pequeno; ser simples de utilizar; ser baseado em componentes de alto nível para construção de interfaces; oferecer uma camada simples e ágil para acesso a dados; implementar padrões de projeto. Vários padrões de projeto foram utilizados na concepção do framework, o que eu devo ao *Gamma and the gang of four* e também ao Martin Fowler pelas excelentes obras sobre Design Patterns. Então você dificilmente vai ver código HTML ou SQL dentro de uma aplicação escrita com o Adianti Framework.

Bom, o papo está bom, mas acredito que você já sabe o suficiente para mergulhar no Adianti Framework por meio da leitura dos capítulos que compõem esta obra. Divirta-se e não deixe de me escrever para relatar suas experiências.

Pablo Dall'Oglio

pablo@dalloglio.net

@pablodalloglio

linkedin.com/in/pablod

fb.com/pablodalloglio

CAPÍTULO 1

Introdução

Este capítulo apresenta os conceitos fundamentais presentes no Adianti Framework. Aqui você conhecerá as premissas de desenvolvimento do framework, suas principais características, sua arquitetura, estrutura de diretórios e como se dá o fluxo de execução de seu funcionamento.

1.1 Características

Aqui veremos as características do Adianti Framework.

Foco em aplicações de negócio

Durante a construção do Adianti Framework, não houve a intenção de criar um Framework generalista, possibilitando o desenvolvimento de qualquer aplicação com ele, como um site ou um blog. Apesar de ser possível realizar tais atividades com o Adianti Framework, ele nasceu dentro de um contexto de desenvolvimento de sistemas de gestão, sistemas de informação, mais popularmente conhecidos como ERP's. Ele não prima pela flexibilidade de configuração visual de seus componentes. Por outro lado, ele apresenta componentes bem elaborados, de alto nível, robustos, prontos e fáceis de serem utilizados para a criação de aplicações de negócio.

Desejamos que o desenvolvedor que utiliza o Adianti Framework consiga criar as interfaces do sistema de maneira ágil, escrevendo um código-fonte limpo, orientado a objetos, elegante e de fácil manutenção. Queremos que o desenvolvedor tenha tempo para focar nas regras de negócio, não em detalhes de implementação, para tal, reunimos as principais bibliotecas existentes para criar bons componentes.

Baseado em padrões

Todo o Adianti Framework é baseado em padrões de projeto. Ao todo, ele utiliza em sua implementação mais de 20 padrões de projeto. Dentre eles, podem ser citados: Factory Method, Singleton, Registry, Strategy, Decorator, Active Record, Identity Field, Foreign Key Mapping, Association Table Mapping, Class Table Inheritance, Composite, Query Object, Layer Supertype, Repository, Model View Controller, Front Controller, Template View, Remote Facade, Lazy Initialization, dentre outros.

Qualidade de código

Todo o framework foi desenvolvido seguindo rígidas regras de escrita de código-fonte. Para provar tal característica, as aplicações de exemplo desenvolvidas no framework podem rodar com o nível máximo de erros habilitado no php.ini e nenhuma mensagem de Warning ou Notice será exibida em tela. Além disso, você não verá a presença de nenhuma “@” (usada para suprimir erros) no código-fonte do framework, muito menos a utilização de recursos como variáveis globais.

Orientado a componentes

Diferentemente de outros Frameworks que levam o desenvolvedor a mesclar HTML e PHP em templates, no Adianti Framework esta será uma situação de exceção. A ideia do Framework é construir sempre que possível interfaces por meio de componentes. Caso você precise de uma funcionalidade visual muito específica, então use o mecanismo de templates do Framework, que permite utilizar HTML para compor uma interface. Não gostamos de escrever código HTML, CSS, jQuery, Bootstrap, dentre outras tecnologias dentro do código-fonte da aplicação. Estes códigos devem ficar escondidos da aplicação, encapsulados dentro de componentes. Assim, o desenvolvedor não precisa ser grande conhecedor destas tecnologias para construir interfaces avançadas, basta conhecer os componentes. Como exemplo, para criar componentes utilizados em um formulário, declaramos da seguinte maneira:

```
<?php  
$nome      = new TEntry('nome');  
$senha     = new TPassword('senha');  
$nascimento = new TDate('nascimento');  
$genero    = new TCombo('genero');  
$curriculo  = new TText('curriculo');  
$estilo     = new TRadioGroup('estilo');  
$habilidades = new TCheckGroup('habilidades');
```

Pequeno e fácil de instalar, e open source

Todo o Framework foi concebido para ser pequeno e de fácil instalação. Desta forma, rodar um aplicativo que utiliza o Adianti Framework será tão simples quanto descompactá-lo, uma vez que você já tenha o ambiente com os pré-requisitos configurados. Não existem pré-requisitos escondidos ou proprietários para rodar uma aplicação feita com o Adianti Framework.

Abstração de base de dados

O Adianti Framework privilegia a independência da aplicação em relação ao banco de dados, mediando esta comunicação por meio de uma camada de abstração que permite desenvolver aplicações complexas sem escrever nenhuma linha de SQL, deixando esta tarefa para o framework. Para tal, são criadas classes para representar as tabelas de um banco de dados, seguindo o padrão de projeto Active Record. Com isso, a mesma aplicação poderá rodar sobre diferentes bancos de dados sem necessidade de reescrita. Além disso, padrões de segurança já implementados no framework evitam ataques como SQL Injection.

```
<?php  
$customers = Customer::where('gender', '=', 'M')  
    ->where('name', 'like', 'A%')  
    ->load();  
  
foreach ($customers as $customer)  
{  
    print $customer->phone;  
}
```

Templates para a construção de sistemas

O framework fornece uma arquitetura para criação de novos sistemas. Com ele diversas atividades como a manipulação de base de dados e a construção de interfaces serão muito mais ágeis. Porém ao criarmos um novo sistema também precisamos de funcionalidades pré-definidas tais como: controle de login, controle de permissões, armazenamento de logs, dentre outros. Você poderia, com base no framework, implementar o seu próprio controle de login e de permissões. Porém, como essas funcionalidades são relativamente comuns, deixamos tais recursos prontos para você ganhar ainda mais velocidade no início de seu projeto. Tais funcionalidades não estão implementadas diretamente no Framework, mas estão presentes nos templates. Um template é uma estrutura pré-definida sobre o framework que contém funcionalidades para acelerar ainda mais a criação da aplicação. Podemos destacar o controle de login, controle de permissão de acesso, montagem de menus, registro de logs de acesso, registro de logs de SQL e registro de logs de inserção, alteração e exclusão de registros, bem como a comunicação entre usuários, e o compartilhamento de documentos.

Um template contém o próprio Adianti Framework e além disso, fornece vários ingredientes adicionais pré-definidos. O template da aplicação em si contém programas prontos, classes de modelo, base de dados (para as permissões e logs), elementos em HTML e CSS. No site do Framework são disponibilizados alguns templates básicos, mas você possui toda a liberdade de criar um template adaptado às suas necessidades, ou mesmo alterar o visual dos templates disponibilizados, ajustando cores, fontes, e posições de elementos.

Toda a aplicação que rodará dentro do layout é escrita somente com os conhecimentos do Framework, utilizando os seus componentes. Na figura a seguir temos um exemplo de um dos temas do Template disponibilizado no site.

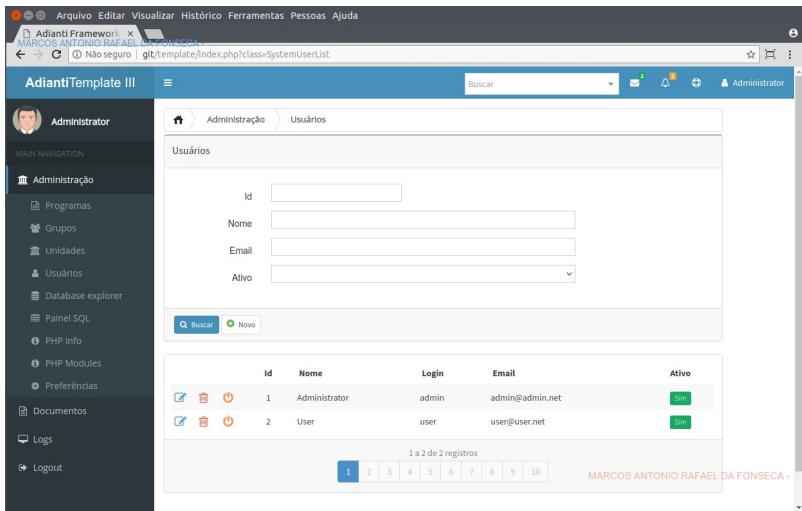


Figura 1 Aplicação rodando com um Template

Obs: Existem templates baseados em Bootstrap e Material design, dentre outros. Você pode utilizar um destes e adaptar às suas necessidades, ou criar um totalmente novo.

1.2 Arquitetura

Uma aplicação criada com o Adianti Framework normalmente será dividida em classes com diversos papéis, dentre os quais podemos citar:

- Model:** Uma entidade do modelo. Estas entidades manipulam dados e desempenham algumas regras de negócio. São representadas por classes armazenadas em `app/model`. Ex: `Cliente`, `Venda`, `Pedido`;
- View:** Interface visual na fronteira entre o sistema e o usuário. Pode ser representada por um Template HTML (`app/resources`), ou um grupo de objetos compondo um objeto maior (`app/view`). Estes objetos geralmente lidam com aspectos de apresentação ou coleta de informações ao usuário final;
- Controller:** Responsável por receber ações vindas de uma classe View e tomar ações. Coordena a sequência de atividades em resposta a uma ação. Para isto, geralmente interage com vários objetos Model para oferecer uma resposta ao objeto View;
- Service:** Responsável por prestar um serviço para a aplicação (serviço interno), como, por exemplo, processar uma regra de negócio complexa, ou prestar um serviço para outra aplicação (ex: REST Service). Representada por uma classe armazenada na pasta `app/service`.

1.3 Estrutura de diretórios

A tabela a seguir apresenta a estrutura de diretórios utilizada pelo Framework.

Tabela 1. Estrutura de diretórios

Pasta	Conteúdo
app	Contém a aplicação desenvolvida
config	Arquivos de configuração da aplicação e do BD
control	Classes controladoras da aplicação
database	Arquivos de banco de dados (Ex: Sqlite)
images	Imagens da aplicação
lib	Bibliotecas e componentes específicos da aplicação
model	Classes de modelo da aplicação (entities)
output	Arquivos temporários gerados (relatórios)
reports	Arquivos criados pelo Designer de documentos
resources	Fragmentos de HTML para usar em templates
service	Serviços disponibilizados pela aplicação
templates	Templates da aplicação (Layout)
view	Classes de apresentação reutilizáveis
lib	Bibliotecas
adianti	Adianti Framework
base	Classes controladoras padrão
control	Classes que gerenciam o fluxo de controle
core	Núcleo do Framework (execução, carga, etc)
database	Camada de acesso a banco de dados
http	Classes para realizar requisições HTTP
include	Arquivos incluídos pelo Framework (js, css)
log	Classes de log de operações
registry	Classes para controle de registro (sessão, cache)
service	Serviços Back-end para componentes
util	Classes utilitárias
validator	Validadores de formulários
widget	Biblioteca de componentes de apresentação
wrapper	Wrappers para componentes complexos
bootstrap	Biblioteca Bootstrap utilizada pelo framework
independent	Bibliotecas Javascript independentes
jquery	Biblioteca jQuery utilizada pelo framework
vendor	Bibliotecas instaladas pelo Composer
</> cmd.php	Utilitário para execuções em linha de comando
</> download.php	Script auxiliar para download de arquivos
</> engine.php	Motor de execução do framework
</> index.php	Ponto de entrada da aplicação Web
</> init.php	Carregamento inicial de classes do Framework.
</> menu.xml	Estrutura do menu da aplicação
</> rest.php	Servidor REST embarcado.

A estrutura é bastante simples e intuitiva. Todas as classes do framework, arquivos incluídos, classes, imagens, etc., estão localizados sob o diretório `/lib/adianti`. O diretório `/lib` ainda possui os diretórios `jquery`, `bootstrap`, dentre outras bibliotecas utilizadas internamente pelo framework. É desejável que o conteúdo do diretório `/lib` não seja alterado, para facilitar a atualização para futuras versões do framework.

O diretório `/app` contém toda a aplicação desenvolvida. Este diretório já possui por convenção uma estrutura pré-determinada onde as classes da aplicação devem ser salvadas. Ao longo do livro conheceremos melhor cada uma destas pastas.

O diretório `/vendor` contém as bibliotecas instaladas pelo Composer, um gerenciador de pacotes de bibliotecas de software para PHP.

1.4 Projetos Adianti

No site da Adianti, você encontrará uma série de projetos. A seguir, explicamos a função de cada um deles nesse ecossistema:

1. **Adianti Framework:** Sempre que utilizarmos este nome, estamos nos referindo ao Framework puro, suas bibliotecas e classes. O Framework puro não vem com uma série de controles como login, permissões, mensagens entre usuários, logs da aplicação. Estes controles fazem parte do Template (a seguir).
2. **Adianti Template:** O Template trata-se do Framework acrescido de uma série de controles, como login, permissões, mensagens, logs, gestão de documentos, e muitos outros. Normalmente, o desenvolvedor utiliza o Template, não o Framework para construir sistemas, pois este já possui muitos controles prontos. O template já vem com diferentes temas como Bootstrap e Material Design.
3. **Adianti Tutor:** O tutor é, como o nome diz, uma aplicação que visa ensinar o uso do Framework. Ele é formado por um grande conjunto de exemplos de utilização de Formulários, Datagrids, e outros.
4. **Adianti Studio:** O Studio é uma IDE Desktop offline, cujo objetivo é oferecer alguns geradores simples e rápidos de código-fonte para programas comuns como formulários de cadastro, datagrids e relatórios. Ele lê definições das tabelas do banco de dados e gera o código-fonte.
5. **Adianti Builder:** O Adianti Builder é um ambiente RAD online em nuvem que possui funcionalidades de alto nível para criação de sistemas. Boa parte da criação da aplicação, desde a modelagem, até o desenho de telas, se dá de maneira visual, utilizando o clique e arraste. Ao final, o desenvolvedor pode realizar download do projeto ou publicá-lo diretamente em um servidor. Além do ambiente de desenvolvimento, o Builder conta com suporte especializado.

1.5 Diferenciais do Adianti Framework

Quando você utiliza a palavra “Framework” no nome, é natural que surjam comparações com outros Frameworks ou outras ferramentas do gênero. Atualmente existem muitas opções, o que dificulta a escolha. Portanto, resolvemos apontar alguns elementos que o diferenciam dos demais.

Essencialmente, a maioria dos Frameworks buscam oferecer uma arquitetura formada por um conjunto de classes para automatizar atividades comuns do dia a dia, geralmente relacionadas ao fluxo de execução de rotinas (rotas), acesso à base de dados, dentre outros. Neste aspecto o Adianti Framework segue a mesma linha dos demais.

As diferenças iniciam pela vocação do Framework para a construção de sistemas de gestão (famosos ERP). Tendo essa vocação definida e uma comunidade formada ao seu redor, todas as iniciativas da Adianti são no sentido de facilitar a construção deste tipo de sistema, oferecendo recursos de alto nível previamente disponíveis para que os desenvolvedores possam focar cada vez menos em detalhes de tecnologia e infraestrutura, e possam se concentrar na especialidade de suas aplicações, nas suas regras de negócio.

Dessa forma, o Template oferece uma gama de funcionalidades prontas para desenvolvedores de sistemas, como gestão de usuários, grupos, multi unidade, permissões, gestão de documentos, gestão de mensagens e notificações, logs de acessos, logs de SQL, logs de mudanças de registros, e muitas outras funcionalidades importantes que o desenvolvedor não precisará desenvolver, pois herdará estas funções prontas para sua aplicação.

Além disso, o Framework conta com um conjunto de componentes de interface de alto nível para construção de sistemas, que fazem com que o desenvolvedor consiga cumprir seu trabalho com a menor quantidade de linhas de código-fonte possível. Os componentes do Framework são completos, pois envolvem implementação tanto para Front-end quanto Back-end, de maneira integrada ao Template. Os componentes do Framework possuem implementações que envolvem código-fonte PHP, Javascript, e uma camada de estilo CSS. Assim, o que muitas vezes é implementado com uma ou duas linhas de código no Framework, necessitariam dezenas ou centenas de linhas de código por parte do desenvolvedor caso escolhesse utilizar um Framework “genérico”, além de demandar conhecimento específico não somente de PHP, como de Javascript e outras bibliotecas específicas, como para montagem de calendários, gráficos, Kanban, etc.

Outro benefício ao utilizar o Framework, é a evolução do conjunto da obra, mantendo a compatibilidade reversa. A Adianti lança novas versões, não somente do Framework, mas do Template, de maneira a preservar a compatibilidade reversa, ou seja, fazendo com que códigos já desenvolvidos em versões anteriores continuem funcionando nas novas com o menor esforço possível de atualização. Isto é muito importante no

desenvolvimento de sistemas, uma vez que é muito caro para as empresas jogarem o trabalho de muitos anos fora, e recomeçar em função de uma nova versão.

A Adianti mantém a evolução do conjunto da obra sem custos de atualização, uma vez que tanto o Framework, quanto o Template são de código aberto.

Outro diferencial, é que a Adianti possui ferramentas de produtividade de alto nível (Studio e Builder), que estão totalmente em sintonia com o Framework e Template, e permitem a acelerar o desenvolvimento e a geração de código-fonte limpo e orientado a objetos. Diferentemente de outras ferramentas RAD que muitas vezes geram um código-fonte poluído de difícil manutenção. As aplicações geradas pelo Studio e pelo Builder não possuem componentes proprietários, não possuem “amarras”, ou seja, o código-fonte é totalmente puro e limpo, permitindo manutenção em qualquer editor de código, e por qualquer profissional PHP, o que é essencial na evolução de um produto.

Ao utilizar o Adianti Framework e Template, você terá produtividade maior do que ao utilizar Frameworks “genéricos”, pois o Adianti Framework/Template trazem uma série de funcionalidades de infraestrutura, e componentes de alto nível prontos, que de outra forma necessitariam conhecimento especialista não somente em PHP, mas também em Javascript, CSS, HTML, jQuery, Bootstrap, e uma série de bibliotecas especialistas, como Date picker, time picker, color picker, calendar, HTML editor, dentre outras.

Além de oferecer componentes visuais, o Framework/Template também oferecem uma camada de alto nível para manipulação de dados, dispensando o desenvolvedor de poluir a aplicação com muitos “SQL” espalhados na aplicação. Com poucas linhas é possível implementar grande parte das operações de CRUD necessárias.

1.6 Começar pelo Framework ou pelo Template?

No site da Adianti, você encontrará para download o Framework e o Template. O link para download do Framework contém o Adianti Framework puro, somente com a biblioteca de classes original e uma interface mínima com menu e instruções iniciais para uso. O Template, por sua vez, contém além do Framework, temas completos e controles de permissão de acesso, login, criação de conta, reset de senha, gestão de usuários, grupos, unidades, controles para uso multiunidade, tradução, logs, comunicação entre usuário, sistema de notificações, gestão e compartilhamento de documentos, e muitos outros. O Template será abordado ao final deste livro.

Se você está começando a aprender o Framework, recomendamos criar os exemplos para testes primeiramente no Framework puro, para posteriormente criar os programas dentro do Template. Ao utilizar o Template, será necessário liberar permissões e cadastrar usuários para poder fazer uso do programa. Já no Framework puro, todos os programas são públicos, o que acelera o aprendizado em um primeiro momento. O Template é indicado para uso depois que você já aprendeu a utilizar o básico do Framework. Utilize o Template quando você já superou a experimentação inicial está decidido e confiante para começar o desenvolvimento do sistema.

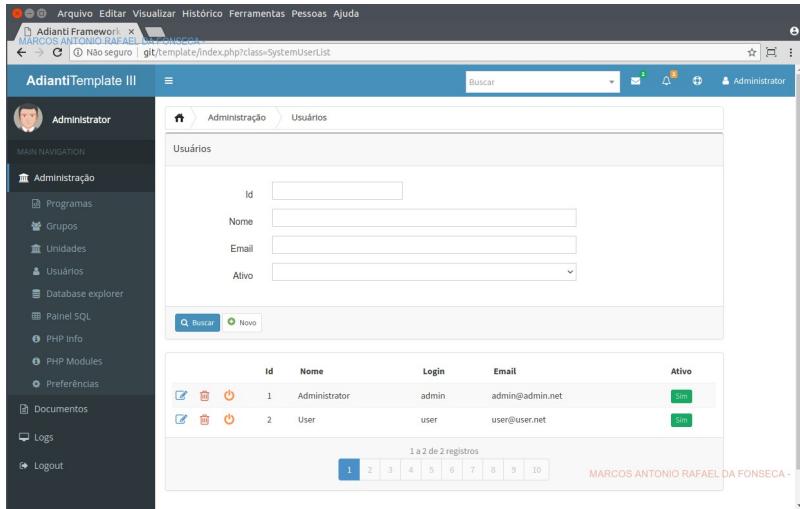


Figura 2 Adianti Template

1.7 Ganhar produtividade com o Studio ou Builder?

Após aprender o essencial da utilização do Framework é natural que você queira acelerar o desenvolvimento. E para tal, existem duas opções: O Studio e o Builder.

O Adianti Studio utiliza tecnologia Desktop Offline, e é uma IDE convencional para PHP com alguns wizards (assistentes passo a passo) para gerar código-fonte de formulários e datagrids simplificados com base na leitura das tabelas do banco de dados. O código gerado é mais bruto e para a maioria dos ajustes você precisará conhecer bem o Framework para fazer manualmente.

O Adianti Builder é uma ferramenta RAD online para construção de aplicações, que vai desde a modelagem visual do banco de dados, até a construção visual de telas para formulários, datagrids, relatórios, documentos, gráficos, calendários e outros. Permite realizar muitas definições de maneira visual, e também conta com pontos de edição para personalização de código-fonte. Ele tem controles mais refinados e é uma ferramenta mais sofisticada, você clica, arrasta e configura o comportamento dos componentes, e ainda tem assistentes que te ajudam a gerar trechos de código com ações específicas.

O Studio é mais voltado para edição de código para quem domina o Adianti Framework, e que prefere fazer as coisas na "mão", com auxílio de alguns assistentes. O Builder é voltado para todos os públicos, desde o iniciante, até o avançado, por permitir realizar uma grande parte das operações de maneira visual, e também dispor de um editor de código-fonte online. Enquanto no Studio você trabalha somente local, no Builder pode trabalhar de qualquer computador, e a qualquer momento, realizar o teste online da aplicação, realizar seu Download, ou publicação em um servidor próprio.

1.8 Um controlador de página

Uma página é representada por uma classe de controle, que por sua vez pode utilizar em seu interior componentes do Framework (como formulários, e datagrids), ou mesmo a renderização de conteúdo HTML.

As classes de controle podem ser filhas de `TPage` ou de `TWindow`. As controladoras filhas de `TPage` são exibidas no quadro central do layout e as filhas de `TWindow` são sempre exibidas em uma nova janela. Existem ainda as cortinas laterais, vistas mais adiante.

A classe a seguir tem como função somente exibir uma imagem na página, o que é realizado pela criação de um objeto `TImage`. Logo em seguida este objeto é adicionado à página pelo método `add()`. Para criar layout's mais elaborados usamos containers como tabelas, divs, frames, caixas, e outros, que permitem empilhar elementos.

`app/control/WelcomeView.class.php`

```
<?php
class WelcomeView extends TPage
{
    public function __construct()
    {
        // construtor da classe-pai
        parent::__construct();

        // instancia a imagem
        $image = new TImage('app/images/frontpage.png');

        // adiciona a imagem à página.
        parent::add($image);
    }
}
```

1.9 A página de entrada

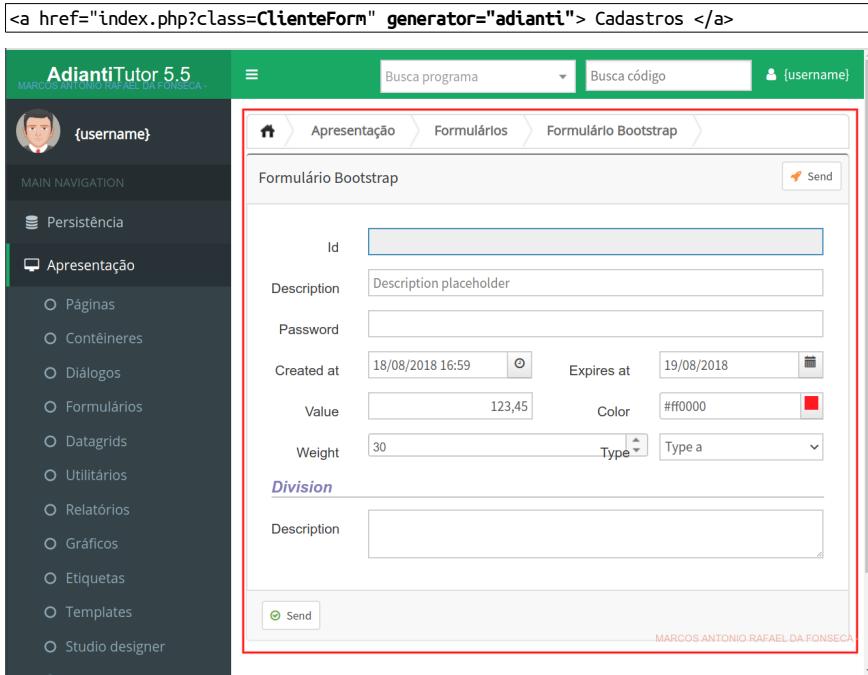
Toda aplicação feita no Adianti Framework possui um ponto de entrada, que é representado pelo `index.php` que realiza a abertura e apresentação inicial da aplicação.

Quando uma aplicação feita no Adianti Framework é carregada pelo navegador, acessamos o `index.php`. Neste momento todo o layout é carregado. O layout é fornecido pelo template e contém toda a decoração da aplicação: menus, barras de navegação, estilos, bibliotecas, imagens, dentre outros. Geralmente ao centro do layout é reservado um local para a página ser renderizada. Na próxima figura, é demonstrado um layout e ao centro está em destaque um quadro contendo um formulário. Este formulário é um conteúdo gerado dinamicamente pelo Framework. Existem duas maneiras de carregar um conteúdo dinâmico pelo Framework. A maneira mais simples é indicando pela URL a classe que desejamos carregar. Neste caso a seguir, estamos indicando claramente a classe controladora que desejamos carregar.

<code>http://localhost/framework/index.php?class=ClienteForm</code>

A segunda forma de efetuar o carregamento de uma página é por meio de um link interno na página que possua um atributo `generator="adianti"` no formato indicado a seguir. O framework automaticamente intercepta links com o atributo `generator="adianti"` e carrega a página no quadro central do layout. A carga é efetuada de maneira transparente pelo back-end do framework: `engine.php`.

Depois da carga inicial do layout, o Framework não mais “recarrega” toda a página em requisições posteriores. Somente o núcleo da página é injetado automaticamente sempre que o usuário navega, clica em opções do menu, em botões de edição, e outros. Este comportamento pode ser percebido pela fluidez na navegação.



The screenshot shows the Adianti Tutor 5.5 application interface. On the left, there is a dark sidebar with a user profile icon and the text '{username}'. Below it is a 'MAIN NAVIGATION' section with several items: Persistência, Apresentação (which is expanded), Páginas, Contêineres, Diálogos, Formulários, Datagrids, Utilitários, Relatórios, Gráficos, Etiquetas, Templates, and Studio designer. The 'Apresentação' item has a red border around its sub-items. The main content area has a light green header with 'Busca programa' and 'Busca código' fields, and a user dropdown. Below the header is a breadcrumb navigation: 'Apresentação > Formulários > Formulário Bootstrap'. The central part of the screen displays a 'Formulário Bootstrap' template. This template includes input fields for 'Id' (with placeholder 'Description placeholder'), 'Description', 'Password', 'Created at' (with date '18/08/2018 16:59' and a calendar icon), 'Expires at' (with date '19/08/2018'), 'Value' (with value '123,45'), 'Color' (with color '#ff0000'), 'Weight' (with value '30'), and 'Type' (with dropdown options 'a' and 'b'). Below these fields is a section titled 'Division' with a 'Description' input field. At the bottom of the form is a 'Send' button with a green checkmark icon. The entire form area is enclosed in a red rectangular border.

Figura 3 Template com o conteúdo em destaque ao centro

1.9.1 O index

Quando abrimos uma aplicação Adianti Framework, o arquivo `index.php` é interpretado. Isto ocorre somente na primeira vez em que o usuário acessa o sistema ou sempre que o mesmo forçar a recarga da página (F5 ou CTRL+R). A recarga ocorre somente nestas situações, por que toda a navegação posterior (clique em opções do menu, postagem de formulários, edição de registros) recarrega somente o “núcleo” da página, não todo o seu conteúdo. Isto ocorre por que as ações são executadas pelo `engine.php`, não pelo `index.php`. O engine é responsável pela execução da classe solicitada e pelo retorno do conteúdo processado, que é automaticamente injetado no “núcleo” da página.

A seguir apresentamos o `index.php`, página de entrada do Framework puro. Ele inicia com a carga do `init.php`, que por sua vez realiza a inicialização do framework, com o carregamento das classes, e definição de configurações. Em seguida é iniciada uma nova seção. `TSession` é a classe responsável por manipular a sessão.

O layout da aplicação é lido a partir do diretório `app/templates`. Como a aplicação pode ter vários temas, o tema oficial é lido do arquivo `application.ini`, que é disponibilizado automaticamente pela variável `$ini`, processada no `init.php`. O layout é lido a partir do arquivo `layout.html` e seu resultado armazenado na variável `$content`.

O menu da aplicação fica armazenado no arquivo `menu.xml`, que contém toda a estrutura do menu em XML. A classe `AdiantiMenuBuilder` faz a leitura deste arquivo e a sua conversão para HTML, pra que o mesmo seja injetado no layout, no lugar da marcação `{MENU}`, que está lá somente guardando seu lugar. O resultado do menu é armazenado na variável `$menu_string` e esta é injetada em seguida dentro do layout.

As bibliotecas necessárias para funcionamento do Framework e aplicação ficam armazenadas no `libraries.html` e este conteúdo é injetado no lugar da marcação `{LIBRARIES}` presente dentro do layout. A substituição é realizada pelo `str_replace()`.

O conteúdo do Template é apresentado em tela por meio do comando `echo`. Ao final, é verificado se o parâmetro “`class`” está presente na URL. Caso afirmativo, a classe correspondente é carregada por meio do método `AdiantiCoreApplication::loadPage()`. Este método executa a classe e injeta seu conteúdo no núcleo do layout.

`index.php`

```
<?php
require_once 'init.php';
$theme = $ini['general']['theme'];
new TSession;

$content = file_get_contents("app/templates/{$theme}/layout.html");
$menu_string = AdiantiMenuBuilder::parse('menu.xml', $theme);
$content = str_replace('{MENU}', $menu_string, $content);
$content = ApplicationTranslator::translateTemplate($content);
$content = str_replace('{LIBRARIES}', file_get_contents("..../libraries.html"), ...);
$content = str_replace('{template}', $theme, $content);
$content = str_replace('{MENU}', $menu_string, $content);
$css = TPage::getLoadedCSS();
$js = TPage::getLoadedJS();
$content = str_replace('{HEAD}', $css.$js, $content);

echo $content;

if (isset($_REQUEST['class'])) {
    $method = isset($_REQUEST['method']) ? $_REQUEST['method'] : NULL;
    AdiantiCoreApplication::loadPage($_REQUEST['class'], $method, $_REQUEST);
}
```

Obs: O index do Template completo é maior, e será explicado posteriormente.

1.9.2 O init

O `init.php` é o script responsável pelo carregamento do Framework, envolvendo carregamento das classes e definições de constantes e de idioma para tradução.

Em primeiro lugar, definimos qual será o autoloader (`spl_autoload_register`), ou seja, qual método será responsável pela carga das classes do framework. Neste caso, é indicada a classe `AdiantiCoreLoader` para realizar a carga das classes do framework. As classes serão carregadas dinamicamente pelo método `autoload()` dessa classe.

O require para `vendor/autoload.php` permite carregar e disponibilizar para a aplicação, bibliotecas instaladas pelo Composer, localizadas na pasta `vendor`.

Em seguida, configurações são lidas do arquivo `application.ini`. Estas configurações são armazenadas na variável `$ini`. Então, é definida a timezone do PHP, por meio da função `date_default_timezone_set()` e é definido o idioma da aplicação por meio do método `setLanguage()` das classes `AdiantiCoreTranslator` (mensagens do núcleo do framework) e `ApplicationTranslator` (mensagens da aplicação). A classe `AdiantiApplicationConfig` carrega as configurações e permite que estas sejam obtidas em qualquer momento da execução, pois esta classe disponibiliza o método `get()` para isto.

Em seguida, são definidas algumas constantes utilizadas pelo Framework. A constante `APPLICATION_NAME` contém o nome da aplicação e é utilizada para separação das variáveis de sessão de duas aplicações, mesmo rodando sob o mesmo domínio. O seu conteúdo deve ser alterado no arquivo `application.ini` (variável `application`). A constante `OS` contém o sistema operacional, `PATH` contém o caminho para o diretório principal da aplicação, e `LANG` contém o idioma utilizado.

`init.php`

```
<?php
// define the autoloader
require_once 'lib/adianti/core/AdiantiCoreLoader.php';
spl_autoload_register(array('Adianti\Core\AdiantiCoreLoader', 'autoload'));
Adianti\Core\AdiantiCoreLoader::loadClassMap();

$loader = require 'vendor/autoload.php';
$loader->register();

// read configurations
$ini = parse_ini_file('app/config/application.ini', true);
date_default_timezone_set($ini['general']['timezone']);
AdiantiCoreTranslator::setLanguage( $ini['general']['language'] );
ApplicationTranslator::setLanguage( $ini['general']['language'] );
AdiantiApplicationConfig::load($ini);
AdiantiApplicationConfig::apply();

// define constants
define('APPLICATION_NAME', $ini['general']['application']);
define('OS', strtoupper(substr(PHP_OS, 0, 3)));
define('PATH', dirname(__FILE__));
define('LANG', $ini['general']['language']);
```

1.9.3 O layout

A página de entrada `index.php` faz basicamente a apresentação do template HTML armazenado no arquivo `layout.html`. O desenvolvedor pode renomear este arquivo, desde que também o faça em sua chamada no `index.php`.

Existem alguns aspectos importantes sobre um template. Em primeiro lugar, o template deve ter as marcações `{LIBRARIES}` e `{HEAD}`, pois estas marcações são substituídas em tempo de execução no `index.php`, pelas bibliotecas necessárias pelo Framework, que por sua vez ficam armazenadas no arquivo `libraries.html`.

O template deve ter um `<div>` com o ID `adianti_div_content`. É neste quadro que o conteúdo do sistema (formulários, listagens) é injetado dinamicamente sempre que o usuário navega na aplicação (clique em opções do menu, submissão de formulários, edição de registros). Este conteúdo é processado pelo `engine.php` no lado do servidor.

O template também deve o `<div>` com o ID `adianti_online_content`. Neste elemento, são carregados conteúdos como janelas sobrepostas ao conteúdo principal; `adianti_online_content2` onde são abertas janelas sobre janelas; e `adianti_right_panel`, onde são abertas cortinas laterais (vistas em exemplos mais adiante).

Tomando estes cuidados básicos, o desenvolvedor pode alterar o HTML, acrescentando novos elementos estruturais e também carregar novos estilos CSS.

Obs: Você encontra a versão completa deste arquivo com o download do Framework.

`app/templates/themeX/layout.html`

```
<html>
<head>
    <title>Adianti Framework :: Samples</title>
    {LIBRARIES}
    {HEAD}
</head>
<body>
    <div class="adianti_container">
        <div class="body">
            <div id="menuDiv">
                <div class="tutor-navbar">
                    <div class="tutor-navbar-inner">
                        {MENU}
                    </div>
                </div>
            </div>
            <div id="adianti_div_content" class="content"></div>
            <div id="adianti_online_content"></div>
            <div id="adianti_online_content2"></div>
            <div id="adianti_right_panel" class="right-panel"></div>
        </div>
    </div>
</body>
</html>
```

1.9.4 O menu

Um dos recursos utilizados pelo `index.php` é o menu da aplicação, armazenado no arquivo `menu.xml`. Este arquivo é processado pela classe `AdiantiMenuBuilder` e o resultado disto é um menu hierárquico como pode ser visto na próxima figura.

No capítulo em que abordaremos o Template do Framework, que já acompanha cadastros de usuários, grupos, e permissões de acesso, veremos que o menu da aplicação é filtrado conforme o perfil do usuário logado. Assim, o usuário somente poderá visualizar e acessar as opções que tiver acesso.

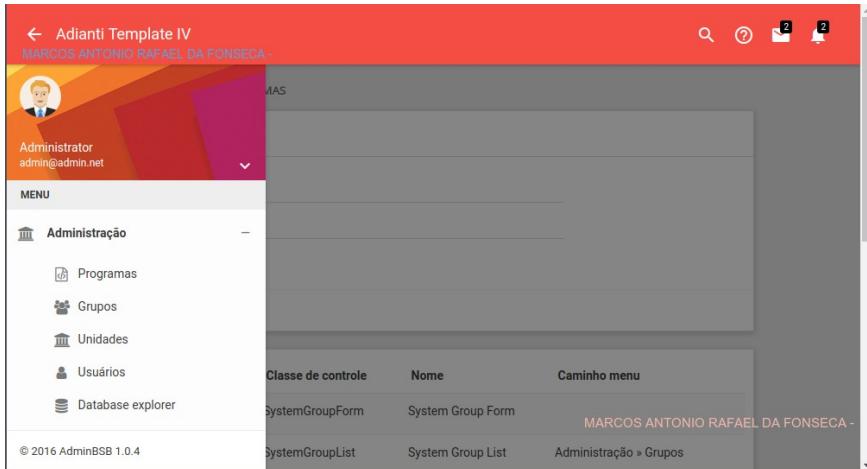


Figura 4 Exemplo de menu

O menu da aplicação é escrito no formato XML como demonstrado a seguir. O menu é renderizado por meio da classe `AdiantiMenuBuilder`, que faz a sua interpretação.

menu.xml

```
<menu>
    <menuitem label='Start here'>
        <menu>
            <menuitem label='Welcome'>
                <menu>
                    <menuitem label='Welcome page'>
                        <icon>app/images/ico_welcome.png</icon>
                        <action>WelcomeView</action>
                    </menuitem>
                </menu>
            </menuitem>
        </menu>
    </menuitem>
</menu>
```

1.9.5 As bibliotecas

O arquivo `libraries.html` contém todas as bibliotecas necessárias para o funcionamento do framework. Ele tem sua estrutura dividida em blocos, que são:

6. Bibliotecas de terceiros: representam bibliotecas JS de terceiros, tais como jQuery, Bootstrap, e plugins específicos, como para seleção de datas (datepicker), e autocomplete de formulários;
7. Bibliotecas do framework: representam as bibliotecas JS do núcleo do framework (`adianti.min.js`) e componentes (`components.min.js`);
8. Javascript da aplicação: caso o desenvolvedor queira, pode utilizar o arquivo `application.js` para armazenar seus Javascript próprios;
9. CSS de terceiros: representam estilos CSS de terceiros, tais como jQuery, Bootstrap, e Font-Awesome, necessários no framework;
10. CSS do framework: representam os estilos CSS do framework (`adianti.min.css`), e de seus componentes (`components.min.css`);
11. CSS da aplicação: estilos próprios e customizados para a aplicação (`application.css`).

`app/templates/themeX/libraries.html`

```
<!-- Bibliotecas de terceiros necessárias pelo framework -->
<script src="lib/jquery/jquery.min.js"></script>
<script src="lib/bootstrap/js/bootstrap.min.js"></script>
<script src="lib/bootstrap/js/bootstrap-plugins.min.js"></script>
<script src="lib/bootstrap/js/locales/bootstrap-datepicker.pt.js"></script>
<script src="lib/jquery/jquery-ui.min.js"></script>
<script src="lib/jquery/jquery-plugins.min.js"></script>

<!-- Biblioteca do núcleo do Adianti framework e componentes -->
<script src="lib/adianti/include/adianti.min.js"></script>
<script src="lib/adianti/include/components.min.js"></script>

<!-- Javascript da aplicação (opcional) -->
<script src="app/lib/include/application.js"></script>

<!-- CSS de terceiros necessários pelo Adianti Framework -->
<link href="lib/jquery/jquery-ui.min.css" type="text/css"/>
<link href="lib/jquery/jquery-plugins.min.css" type="text/css"/>
<link href="lib/bootstrap/css/bootstrap.min.css" type="text/css"/>
<link href="lib/bootstrap/css/bootstrap-plugins.min.css" type="text/css"/>

<!-- CSS do Adianti Framework e de seus componentes -->
<link href="lib/adianti/include/adianti.min.css" type="text/css"/>
<link href="lib/adianti/include/components.min.css" type="text/css"/>

<!-- CSS da aplicação-->
<link href="app/templates/{template}/application.css" type="text/css"/>
```

Obs: Algumas linhas foram suprimidas na edição para ajuste de espaçamento.

1.9.6 O engine

O arquivo `engine.php` é o motor de execução do framework. Todas requisições de páginas passam por ele. Quando o usuário entra no sistema pela primeira vez, o `index.php` solicita o carregamento da página por meio do método `AdiantiCoreApplication::loadPage()`, que por sua vez, faz uma requisição para o `engine.php`, realizando o carregamento da página. Além disso, qualquer clique do usuário sobre algum link que tenha a tag `generator="adianti"`, é automaticamente “sequestrado” pelo framework, que por sua vez, executa a função `_adianti_load_page()`, realizando a carga do `engine.php`, via Ajax. Essa sistemática, faz com que a navegação do sistema seja rápida.

Basicamente no início do `engine.php`, é requisitado script `init.php`, que por sua vez, realiza o carregamento do framework, suas classes necessárias, e as definições de constantes e de idioma. Em seguida, é declarada a classe `TApplication`, filha de `AdiantiCoreApplication`, que é a classe do núcleo do Framework, responsável por fazer com que ele execute as ações solicitadas pelo usuário por meio da URL.

Caso seja necessário acrescentar um controle de acesso (permissões), este pode ser realizado dentro do método `run()` de `TApplication`, observando se o usuário possui permissão para visualizar a página requisitada. Um controle deste tipo é realizado pelo Template, que implementa um controle de permissões de acesso.

O método `run()` roda a classe solicitada pela URL (`$_REQUEST`), processando todo seu conteúdo, que é capturado e injetado na parte central do layout da aplicação. Este engine apresentado é do Framework puro. O engine do Template é bem mais completo, pois lida com outros aspectos como o controle de permissões de acesso.

`engine.php`

```
<?php
require_once 'init.php';

class TApplication extends AdiantiCoreApplication
{
    public static function run($debug = null)
    {
        new TSession;

        if ($_REQUEST)
        {
            $ini    = AdiantiApplicationConfig::get();
            $debug = is_null($debug)? $ini['general']['debug'] : $debug;
            parent::run($debug);
        }
    }
} TApplication::run();
```

Dentro do arquivo `application.ini`, pode ser encontrado a variável `debug`. Se `debug` estiver ligado, as mensagens de exceção apresentadas terá detalhes técnicos. Quando o sistema estiver em produção, o `debug` deve estar desligado.

[app/config/application.ini](#)

```
[general]
timezone = America/Sao_Paulo
language = pt
application = sample
theme = theme3
debug = 1
```

1.10 Adianti Tutor

Muitos dos códigos abordados neste livro integram uma aplicação chamada Tutor. O Tutor é um pequeno sistema construído com o Adianti Framework que pretende demonstrar todas as suas funcionalidades. Você deve instalá-lo para um melhor acompanhamento das lições presentes neste livro e executar os exemplos a medida em que vai lendo-os. Ele pode ser baixado pelo endereço:

<http://www.adianti.com.br/framework-tutorial>

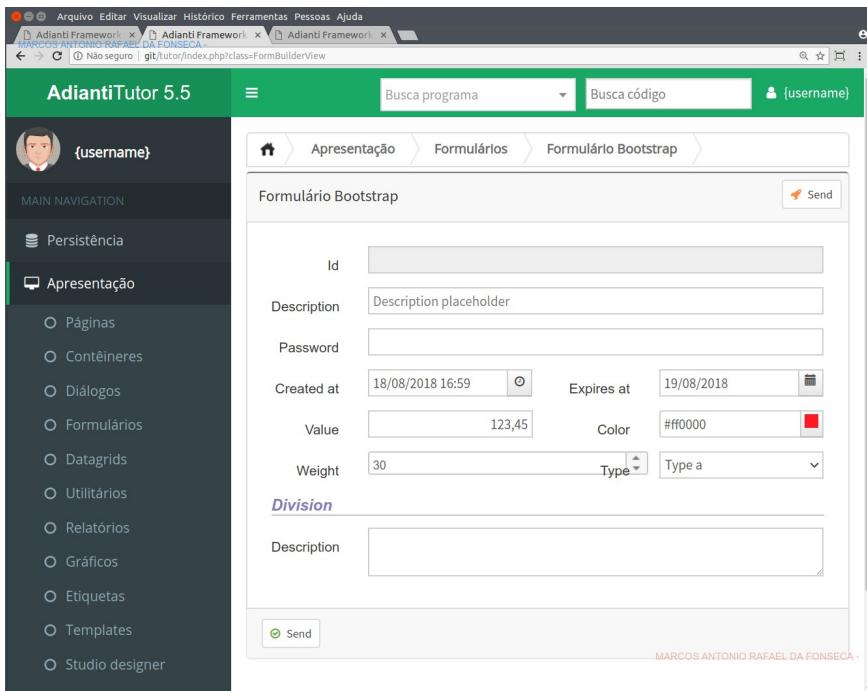


Figura 5 Aplicação tutor

CAPÍTULO 2

Instalação e configuração

Neste capítulo vamos aprender a dar os primeiros passos com o framework. Vamos abordar a instalação dos pré-requisitos de ambiente em Linux, Windows e MacOS, bem como os passos para instalação do Framework.

2.1 Instalação do ambiente

2.1.1 Ambiente Linux

Nesta seção, será abordada a instalação dos pré-requisitos para rodar o Adianti Framework em Linux Ubuntu 18.04. Dessa forma, instalamos o Apache, o PHP, bem como o suporte aos bancos de dados SQLite, PostgreSQL e Mysql. O suporte ao SQLite é fundamental, visto que a maioria das aplicações de exemplo que acompanham o Framework utilizam este banco de dados para distribuir exemplos. A seguir, os passos:

```
sudo su
apt-get update
apt-get install apache2 libapache2-mod-php php php-sqlite3 php-pgsql \
    php-mysql php-xml php-curl php-opcache php-mbstring

a2dismod mpm_event
a2dismod mpm_worker
a2enmod mpm_prefork
a2enmod rewrite
a2enmod php7.2
service apache2 restart
```

Certifique-se de que o Apache tenha habilitado o módulo `rewrite`, para fazer a interpretação dos arquivos `.htaccess`, uma vez que o Framework utiliza este arquivo para proteger o acesso a determinadas pastas com arquivos de configuração, como `app/config`. A seguir, temos um trecho de código que deve estar presente no `apache2.conf` ou `httpd.conf` dentro da cláusula `Directory`, para habilitar esta configuração.

```
AllowOverride All
```

A configuração do PHP é definida pelo arquivo `php.ini`, que no caso do Ubuntu 18.04, fica localizado em `/etc/php/7.2/apache2/php.ini`. Para ambientes de desenvolvimento, recomendamos as cláusulas a seguir, que por sua vez, habilitam todas os tipos de notificação de erros, aumentam o tempo de execução dos scripts, permitem upload de arquivos de até 100 Mb, e aumentam o tempo da sessão para 4 horas.

```
error_log = /tmp/php_errors.log
display_errors = On
memory_limit = 256M
max_execution_time = 120
error_reporting = E_ALL
file_uploads = On
post_max_size = 100M
upload_max_filesize = 100M
session.gc_maxlifetime = 14400
```

Já para ambientes de produção, recomendamos aumentar o nível de tolerância quanto aos tipos de notificações de erros, e desligar a exibição de erros, ao mesmo tempo em que mantemos o registro de logs. As demais cláusulas permanecem inalteradas.

```
display_errors = Off
error_reporting = E_ALL & ~E_DEPRECATED & ~E_STRICT & ~E_NOTICE
```

Obs: Maiores informações em: <https://phpcom.br/installacao-php-linux>

2.1.2 Ambiente Windows

Para preparar o ambiente Web no Windows, utilize um instalador como o Wamp (<http://www.wampserver.com>), ou Xampp (<https://www.apachefriends.org>) que já instala os pré-requisitos como Apache e PHP.

Certifique-se de que o Apache tenha habilitado o módulo `rewrite`, para fazer a interpretação dos arquivos `.htaccess`, uma vez que o Framework utiliza este arquivo para proteger o acesso a determinadas pastas com arquivos de configuração, como `app/config`. A seguir, temos um trecho de código que deve estar presente no `apache2.conf` ou `httpd.conf` dentro da cláusula `Directory`, para habilitar esta configuração.

AllowOverride All

A configuração do PHP é definida pelo arquivo `php.ini`, que no caso do Xampp, fica localizado em `C:\xampp\php\php.ini`, e no caso do Wampp, em `C:\wamp\bin\apache\apache2.x.y\bin\php.ini`. Para ambientes de desenvolvimento, recomendamos as cláusulas a seguir, que por sua vez, habilitam todas os tipos de notificação de erros, aumentam o tempo de execução dos scripts, permitem uploads maiores e sessões mais longas.

```
display_errors = On
memory_limit = 256M
max_execution_time = 120
error_reporting = E_ALL
file_uploads = On
post_max_size = 100M
upload_max_filesize = 100M
session.gc_maxlifetime = 14400
```

Já para ambientes de produção, recomendamos aumentar o nível de tolerância quanto aos tipos de notificações de erros, e desligar a exibição de erros, ao mesmo tempo em que mantemos o registro de logs. As demais cláusulas permanecem inalteradas.

```
display_errors = Off  
error_reporting = E_ALL & ~E_DEPRECATED & ~E_STRICT & ~E_NOTICE
```

Obs: Maiores informações em: <https://phpcom.br/installacao-php-windows>

2.1.3 Ambiente MacOS

Para preparar o ambiente Web no MacOS, utilize um instalador como este (<https://php-osx.liip.ch/>), que instala o PHP com o Apache pré-instalado no MAC.

Certifique-se de que o Apache tenha habilitada a interpretação dos arquivos `.htaccess`, uma vez que o framework utiliza este arquivo para proteger o acesso a determinadas pastas com arquivos de configuração, como `app/config`. A seguir, temos um trecho de código que deve estar presente no `apache2.conf` ou `httpd.conf` dentro da cláusula `Directory`, para habilitar esta configuração.

AllowOverride All

A configuração do PHP é definida pelo arquivo `php.ini`, que no caso deste instalador, fica localizado em `/usr/local/php5/lib/php.ini`. Para ambientes de desenvolvimento, recomendamos as cláusulas a seguir, que por sua vez, habilitam todas os tipos de notificação de erros, aumentam o tempo de execução dos scripts, permitem upload de arquivos de até 100 Mb, e aumentam o tempo da sessão para 4 horas.

```
error_log = /tmp/php_errors.log  
display_errors = On  
memory_limit = 256M  
max_execution_time = 120  
error_reporting = E_ALL  
file_uploads = On  
post_max_size = 100M  
upload_max_filesize = 100M  
session.gc_maxlifetime = 14400
```

Já para ambientes de produção, recomendamos aumentar o nível de tolerância quanto aos tipos de notificações de erros, e desligar a exibição de erros, ao mesmo tempo em que mantemos o registro de logs. As demais cláusulas permanecem inalteradas.

```
display_errors = Off  
error_reporting = E_ALL & ~E_DEPRECATED & ~E_STRICT & ~E_NOTICE
```

Obs: Maiores informações em: <https://phpcom.br/installacao-php-macos>

2.2 Instalação do framework

2.2.1 Ambiente Linux

A instalação do Framework passa inicialmente por seu download que pode ser realizado em <http://www.adianti.com.br/framework-download>.

Após o download do Adianti Framework para Web, o mesmo pode ser descompactado na pasta `/var/www/html` do Ubuntu, que é a pasta padrão do Apache.

```
# cd /var/www/html  
# unzip adianti-framework-<versão>.zip
```

Com este comando, é criada a pasta framework, com todas as classes do framework, bem como a estrutura de uma aplicação vazia (pasta `app`). Você já pode abrir o navegador e digitar <http://localhost/framework> e uma aplicação inicial enxuta será carregada, como pode ser visto na figura a seguir. Automaticamente o ponto de entrada `index.php` será executado. É recomendado posteriormente renomear esta pasta para o nome desejado de sua aplicação.

2.2.2 Ambiente Windows

A instalação do framework passa inicialmente por seu download que pode ser realizado em <http://www.adianti.com.br/framework-download>. Após o download, o mesmo pode ser descompactado na pasta `C:\wamp\www`, caminho padrão para a instalação do Wamp, ou `C:\xampp\htdocs`, caminho padrão para instalação do Xampp.

Ao descompactarmos o arquivo, é criada a pasta framework, com todas as classes do framework, bem como a estrutura de uma aplicação vazia (pasta `app`). Você pode abrir o navegador e digitar <http://localhost/framework> e uma aplicação inicial será carregada. Automaticamente `index.php` é executado.

2.2.3 Ambiente MacOS

A instalação do framework passa inicialmente por seu download que pode ser realizado em <http://www.adianti.com.br/framework-download>.

Após o download do Adianti Framework para Web, o mesmo pode ser descompactado na pasta `/Library/WebServer/Documents/` do MacOS, que é a pasta padrão do Apache.

```
# cd /Library/WebServer/Documents/  
# unzip adianti-framework-<versão>.zip
```

Com este comando, é criada a pasta framework, com todas as classes do framework, bem como a estrutura de uma aplicação vazia (pasta `app`). Você já pode abrir o navegador e digitar <http://localhost/framework> e uma aplicação inicial enxuta será carregada, como pode ser visto na figura a seguir. Automaticamente o ponto de entrada `index.php` será executado. É recomendado posteriormente renomear esta pasta para o nome desejado de sua aplicação.

CAPÍTULO 3

Modelos e persistência

Todo o acesso a bancos de dados dentro do Adianti Framework passa por uma camada orientada a objetos que torna a maioria das operações transparentes ao desenvolvedor. Esta camada de manipulação visa dar maior agilidade na criação das aplicações, bem como fazer com que todo o acesso aos dados ocorra de uma maneira orientada a objetos. Assim, na grande maioria das vezes, você não precisará mais escrever os tradicionais INSERT, UPDATE e DELETE, mas irá simplesmente executar métodos sobre objetos, e internamente o Framework tratará de executar os comandos SQL corretos.

3.1 Modelo utilizado

3.1.1 Modelo de classes

Ao longo deste capítulo, trabalharemos com vários exemplos envolvendo acesso a banco de dados. Mas antes de pensar na estrutura do banco de dados, modelamos a estrutura da aplicação com seus objetos, atributos, métodos e relacionamentos. Para os exemplos deste capítulo, elaboramos um modelo simplificado com as classes: **Customer** (cliente), **Category** (categoria), **City** (cidade), **State** (estado), **Contact** (contato), **Skill** (habilidade), **Sale** (venda), **SaleItem** (item da venda), e **Product** (produto). As relações entre os objetos são as seguintes:

- Um objeto **Customer** (cliente) está associado a um objeto **City** (cidade);
- Um objeto **Customer** (cliente) está associado a um objeto **Category** (categoria). A categoria pode ser: frequente, casual, varejista, etc;
- Um objeto **Customer** (cliente) possui uma composição com nenhum ou vários objetos do tipo **Contact** (contato). Isto significa que um objeto **Contact** (objeto parte) é parte exclusiva somente um objeto **Customer** (objeto todo);

- Um objeto **Customer** (cliente) possui uma agregação com nenhum ou vários objetos **Skill** (habilidade). Isto significa que um objeto **Skill** (objeto parte) pode ser parte compartilhada de diferentes objetos **Customer** (objeto todo);
- Um objeto **City** (cidade) está associado a um objeto **State** (estado);
- Um objeto **Sale** (venda) está associado a um objeto **Customer** (cliente);
- Um objeto **Sale** (venda) possui uma composição com nenhum ou vários objetos **SaleItem** (item da venda). Cada objetos **SaleItem** (parte) será parte exclusiva do objeto **Sale** (todo);
- Um objeto **SaleItem** (item da venda) está associado a um objeto **Product**.

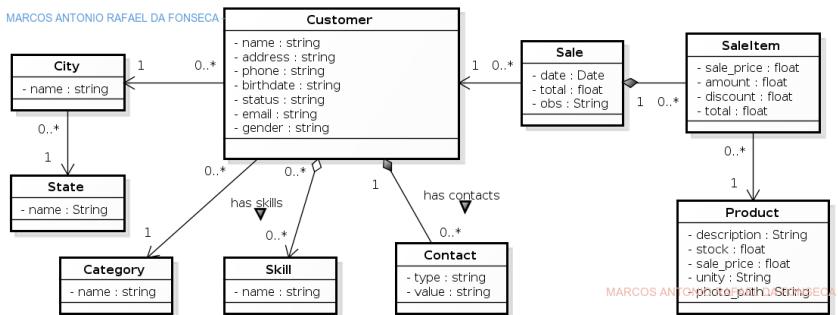


Figura 6 Modelo de classes

Obs: Você encontra maiores informações sobre orientação a objetos em PHP com o livro “*PHP Programando com orientação a objetos*”, do mesmo autor.

3.1.2 Modelo relacional

Um modelo de classes pode ser facilmente convertido em um modelo relacional por meio de técnicas de mapeamento objeto-relacional (M.O.R.). A partir do modelo de classes exposto anteriormente, derivamos um modelo relacional contendo as tabelas para armazenar os objetos da aplicação. As seguintes conversões foram realizadas:

- O objeto **Customer** (cliente) está relacionado com **Category** (categoria) e com **City** (cidade) por meio de chaves estrangeiras (*foreign key mapping*);
- O objeto **City** (cidade) está relacionado com **State** (estado) por meio de chaves estrangeiras (*foreign key mapping*);
- A relação de composição entre **Customer** (cliente) e **Contact** (contato) foi mapeada por meio de uma chave estrangeira (*foreign key mapping*), uma vez que em uma relação de composição, a parte é exclusiva do objeto todo;
- A relação de agregação entre **Customer** (cliente) e **Skill** (habilidade) foi mapeada pela tabela associativa (*association table mapping*) **customer_skill** já que na agregação, a parte não é exclusiva do todo;

- O objeto **Sale** (venda) está relacionado com **Customer** (cliente) por meio de chaves estrangeiras (*foreign key mapping*);
- A relação de composição entre **Sale** (venda) e **SaleItem** (item da venda) foi mapeada por meio de uma chave estrangeira (*foreign key mapping*), uma vez que em uma relação de composição, a parte é exclusiva do objeto todo;
- O objeto **SaleItem** (item da venda) está relacionado com **Product** (produto) por meio de chaves estrangeiras (*foreign key mapping*);

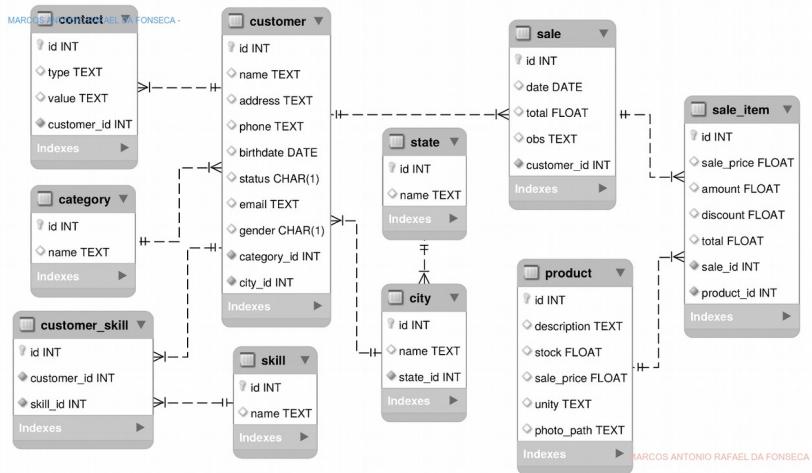


Figura 7 Modelo relacional

3.2 Configuração e acesso ao banco de dados

3.2.1 Criação do banco de dados

O modelo relacional exposto anteriormente é distribuído junto com o código-fonte do tutor no arquivo `app/database/samples.sql`. Além deste, uma base de dados SQLite pronta com dados está no arquivo `app/database/samples.db`. Você pode praticar os exemplos que envolvem base de dados, recriando esta base em outros sistemas gerenciadores como PostgreSQL, ou MySQL se assim preferir, uma vez que o Framework é independente de base de dados como será visto mais adiante.

Obs: A base de dados de exemplos foi suficientemente testada em MySQL, SQLite e PostgreSQL. Para outros bancos de dados, alguns ajustes podem ser necessários.

3.2.2 Configuração do acesso

A configuração de acesso a uma base de dados no Adianti Framework é realizada por meio de arquivos de configuração no formato INI. Em aplicações desenvolvidas no Adianti Framework você não encontrará strings de conexão dentro dos programas com identificação de base de dados, usuários ou senhas.

O primeiro passo para disponibilizar uma base de dados para uma aplicação construída no framework é criar um arquivo INI, onde configuraremos os dados de acesso à base de dados. A partir deste arquivo utilizamos o seu próprio nome quando desejamos estabelecer uma conexão com a base de dados.

O Adianti Framework utiliza a biblioteca PDO para fazer acesso à base de dados, o que em teoria permite que uma aplicação acesse vários tipos de bancos de dados. Até o momento a camada de persistência foi homologada para:

- SQLite;
- MySQL;
- PostgreSQL;
- SQL Server;
- Oracle;
- Firebird.

Obs: Os arquivos de configuração de acesso à base de dados ficam na pasta `app/config`. Cada base de dados deve ter um arquivo INI diferente.

No arquivo a seguir apresentamos a configuração de acesso a uma base em SQLite. Esta será a base utilizada ao longo do livro, e vem com o Tutor. Como SQLite é uma base em sistema de arquivos, não há autenticação por usuário e senha. Além disso, para que a aplicação consiga gravar na base de dados, é necessário que o arquivo `.db`, bem como o diretório no qual ele se encontra, possuam permissão de escrita.

app/config/exemplo-sqlite.ini

```
host      =
name     = "app/database/samples.db"
user      =
pass      =
type      = "sqlite"
prep     = "1" // se utilizará prepared statements
```

A seguir, um exemplo de conexão com base de dados em PostgreSQL.

app/config/exemplo-postgres.ini

```
host = "192.168.1.102"
port = "" // vazio usa a porta padrão (5432)
name = "samples"
user = "postgres"
pass = "postgres"
type = "pgsql"
prep = "1" // se utilizará prepared statements
```

A seguir, um exemplo de conexão com base de dados em MySQL. O atributo “host” pode ser preenchido com o IP da base de dados, ou caminho do socket no formato “localhost:/var/run/mysqld/mysqld.sock”. Caso não saiba o socket, use o IP.

app/config/exemplo-mysql.ini

```
host = "127.0.0.1" // verificar o caminho real do socket
port = "" // vazio usa a porta padrão (3306)
name = "tutor"
user = "root"
pass = "mysql"
type = "mysql"
prep = "1" // se utilizará prepared statements
char = "ISO" // use somente se a base de dados for ISO-8859-1
```

A seguir, um exemplo de conexão com base de dados em SQLServer.

app/config/exemplo-mssql.ini

```
host = "192.168.1.103"
name = "tutor"
user = "sa"
pass = "12345678"
type = "mssql"
```

A seguir, um exemplo de conexão com base de dados em Oracle. Além dos atributos básicos, use o atributo **char** para definir o charset, **flow** para forçar que os atributos sejam mapeados em *lower case*, **date** para definir o formato de data, **time** para o formato de tempo, e **nsep** para definir os separadores numéricos.

app/config/exemplo-oracle.ini

```
host = "192.168.1.103"
port = "1521"
name = "tutor"
user = "system"
pass = "12345678"
type = "oracle"
char = "AL32UTF8" // alterar charset de conexão
flow = "1" // garantir atributos em lowercase
date = "YYYY-MM-DD" // NLS_DATE_FORMAT
time = "YYYY-MM-DD HH:MI:SS.FF" // NLS_TIMESTAMP_FORMAT
nsep = ",." // NLS_NUMERIC_CHARACTERS
```

Também é possível definir a conexão Oracle usando TNS:

app/config/exemplo-oracle-tns.ini

```
user = "system"
pass = "12345678"
type = "oracle"
tns = "(DESCRIPTION =
  (ADDRESS_LIST =
    (ADDRESS = (PROTOCOL = TCP)(HOST = 192.168.0.13)(PORT = 1521))
  )
  (CONNECT_DATA =
    (SERVICE_NAME = exemplos)
  )
)
flow = "1"
```

A seguir, um exemplo de conexão com base de dados em Firebird.

app/config/exemplo-firebird.ini

```
host = "192.168.1.103"
port = ""
name = "c:\teste2.fdb"
user = "sysdba"
pass = "masterkey"
type = "fbird"
```

Obs: Caso seja adicionada a cláusula `prep = "1"` ao arquivo de configuração, todas as *queries* enviadas pelo Framework para o Banco de dados, utilizarão *Prepared Statements*.

O arquivo de acesso a banco de dados pode ser definido também em PHP. Neste caso, basta utilizar a extensão `.php` e um comando `return`, retornando um vetor contendo as mesmas posições já explicadas anteriormente para os arquivos `.ini`. O exemplo a seguir demonstra um arquivo de configuração para SQLite, mas as posições são as mesmas para outros bancos de dados, mudando somente o conteúdo.

app/config/exemplo-sqlite.php

```
<?php
return [
    'host'  => "",
    'port'  => "",
    'name'  => "app/database/samples.db",
    'user'  => "",
    'pass'  => "",
    'type'  => "sqlite",
    'prep'  => "1"
];
```

Para a utilização de arquivos de configuração `.ini`, é importante que o módulo Rewrite do Apache esteja habilitado. Este módulo libera a interpretação dos arquivos `.htaccess` que se encontram em diretórios estratégicos com o `app/config`. Os arquivos `.htaccess` definem regras de acesso à estes diretórios. No caso do Framework, as regras são restritivas, mas elas somente são executadas se o módulo Rewrite estiver habilitado.

Sem este módulo habilitado, estes diretórios ficam desprotegidos e os arquivos `.ini` ficam vulneráveis e suscetíveis a leitura, o que é um problema de segurança. Então a recomendação é sempre habilitar o módulo Rewrite, ou utilizar os arquivos de configuração com a extensão `.php`.

3.2.3 Como executar os exemplos

Para executar os exemplos que veremos ao longo deste capítulo será necessário criar controladores de página. É por meio de um controlador de página que tudo acontece no Framework. Para acessar um controlador de página por meio de seu nome pela Web, basta digitarmos a URL do sistema com este final:

<http://localhost/tutor/index.php?class=ExecuteSample>

Controladores de página serão explicados no capítulo 4, que abordará os componentes visuais e também no capítulo 5, que abordará execução e controle. Por enquanto, para executar os exemplos a seguir, vamos criar um controlador de páginas vazio como o listado a seguir. A maioria dos controladores de páginas é subclasse de `TPage`. Para simplificar, vamos colocar os códigos dentro do método construtor ao redor de um controle de exceções. A classe `TMessage` é responsável por exibir uma caixa de diálogo quando ocorrer alguma exceção.

app/control/dbsamples/ExecuteSample.class.php

```
<?php
class ExecuteSample extends TPage
{
    public function __construct()
    {
        parent::__construct();

        try
        {
            // Aqui vai o código
        }
        catch (Exception $e)
        {
            new TMessage('error', $e->getMessage());
        }
    }
}
```

Obs: Apesar dos exemplos, não é recomendado realizar consultas à base de dados diretamente em um controlador de páginas. O local apropriado é dentro de um Active Record.

3.2.4 Transações e queries manuais

Na grande maioria das vezes em que interagiremos com a base de dados, será por meio de objetos (veremos mais adiante). Mas mesmo assim, é interessante sabermos como realizar consultas de maneira manual na base de dados por meio do Framework. Toda a interação com a base de dados ocorre dentro de uma transação, que é aberta e fechada por meio da classe `TTransaction`. Esta classe abre conexão com uma base de dados configurada na pasta `app/config` e recebe o próprio nome do arquivo como parâmetro do método `open()`.

A classe `TTransaction` possui o método `get()`, que retorna um objeto da classe `PDO` do PHP já instanciado. Então, neste ponto, basta utilizarmos os seus métodos nativos da linguagem, como o método `query()` para realizar uma consulta.

Obs: Não entraremos em detalhes sobre o funcionamento da classe `PDO` por tratar-se de uma classe nativa do PHP. Maiores informações em <http://www.php.net/pdo>.

```
app/control/dbsamples/ManualQuery.class.php


---


<?php
class ManualQuery extends TPage
{
    public function __construct()
    {
        parent::__construct();
        try
        {
            TTransaction::open('samples'); // abre uma transação
            $conn = TTransaction::get(); // obtém a conexão

            // realiza a consulta
            $result = $conn->query('SELECT id, name from customer order by id');

            foreach ($result as $row) // exibe os resultados
            {
                print $row['id'] . '-';
                print $row['name'] . "<br>\n";
            }
            TTransaction::close(); // fecha a transação.
        }
        catch (Exception $e)
        {
            new TMessage('error', $e->getMessage());
        }
    }
}
```

Obs: Após abrir uma transação pelo método `TTransaction::open()`, a qualquer momento podem ser executados os métodos `TTransaction::getDatabase()`, que retorna o nome da conexão ativa, e `TTransaction::getDatabaseInfo()`, que retorna os dados da conexão. Estes métodos podem ser executados inclusive a partir de outras classes.

3.2.5 Queries manuais com prepared statements

Como vimos no exemplo anterior, a intenção ao utilizarmos o Framework não é escrever queries manuais, visto que as classes nativas do Framework farão o trabalho de comunicação com o banco de dados por meio de objetos que gerarão de maneira transparente as instruções SQL de maneira segura.

Porém, eventualmente precisaremos escrever queries complexas para buscar informações no banco de dados que da maneira nativa (orientada a objetos) seria muito complexo ou oneroso. Em casos de exceção nos permitimos escrever queries manuais, desde que, seguindo boas práticas como o uso de *Prepared Statements*. O exemplo a seguir demonstra como escrever uma query em que os parâmetros são injetados na consulta SQL de maneira segura.

No código-fonte a seguir, enquanto o método `prepare()` prepara um SQL para execução posterior, o método `execute()` o executa, recebendo os parâmetros que são “encaixados” no lugar dos símbolos de “?”. Neste exemplo, o método `fetchAll()` retorna todas as linhas encontradas pela consulta.

app/control/dbsamples/ManualPrepareQuery.class.php

```
<?php
class ManualPrepareQuery extends TPage
{
    public function __construct()
    {
        parent::__construct();
        try
        {
            TTransaction::open('samples'); // abre uma transação
            $conn = TTransaction::get(); // obtém a conexão

            $sth = $conn->prepare('SELECT id, name from customer
                WHERE id >= ? AND id <= ?');

            $sth->execute(array(3,12)); // passa os parâmetros
            $result = $sth->fetchAll();

            // exibe os resultados
            foreach ($result as $row)
            {
                print $row['id'] . '-';
                print $row['name'] . "<br>\n";
            }
            TTransaction::close(); // fecha a transação.
        }
        catch (Exception $e)
        {
            new TMessage('error', $e->getMessage());
        }
    }
}
```

3.2.6 Conexão manual

Uma conexão geralmente é aberta por meio do arquivo de configuração de acesso localizado em `app/config`. Outra maneira de abrir uma conexão é definir explicitamente as informações da conexão, como no exemplo a seguir. Esta maneira não é muito recomendada pois, acaba “espalhando” detalhes técnicos de infraestrutura no código-fonte.

app/control/dbsamples/ManualConnection.class.php

```
<?php
class ManualConnection extends TPage
{
    public function __construct()
    {
        parent::__construct();
        try
        {
            // connection info
            $db = array();
            $db['host'] = '';
            $db['port'] = '';
            $db['name'] = 'app/database/samples.db';
            $db['user'] = '';
            $db['pass'] = '';
            $db['type'] = 'sqlite';

            TTransaction::open(NULL, $db); // open transaction
            $conn = TTransaction::get(); // get PDO connection
        }
    }
}
```

3.3 Manipulação de objetos

3.3.1 O padrão Active Record

Todos os objetos da camada de modelo de uma aplicação Adianti Framework são objetos Active Record. Um Active Record é um dos padrões de projeto (*design patterns*) mais conhecidos e implementado na maioria dos frameworks. Conforme Martin Fowler, um objeto Active Record é:

"An object that wraps a row in a database table or view, encapsulates the database access, and adds domain logic on that data."

Um Active Record é um objeto que representa uma linha de uma tabela (ou view), e encapsula ao mesmo tempo o acesso aos dados e a lógica de negócios daquela entidade. Para exemplificar, podemos pensar em um objeto Livro, que pode ter métodos como `registrarEmprestimo()` e `registrarDevolucao()`, que são métodos de negócio e também pode ter métodos como `store()`, `delete()` e `load()` que são métodos de acesso aos dados (persistência).

Geralmente os objetos Active Record não implementam estes métodos de acesso aos dados `store()`, `delete()` e `load()`, uma vez que nossos objetos teriam misturado em um mesmo local, métodos de negócio e de persistência. Para resolver isso, os objetos Active Record generalizam estas operações, que são implementadas por superclasses. No caso do Adianti Framework, os métodos de persistência são implementados pela classe `TRecord`. Assim, todos os objetos de domínio da aplicação devem ser subclasses de `TRecord`, herdando um conjunto de métodos de persistência.

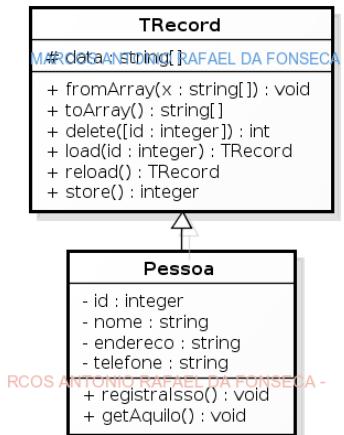


Figura 8 Um objeto Active Record

3.3.2 Definindo uma classe Active Record

O primeiro passo para lidar com objetos é definir uma classe Active Record. A partir dos próximos exemplos, vamos trabalhar com objetos do tipo `Customer` (clientes), então definimos esta classe como filha da classe `TRecord` do Framework. Isto já é o suficiente para termos os métodos de persistência `load()`, `store()` e `delete()`, dentre outros.

Algumas definições devem ser realizadas na forma de constantes da classe. São elas: `TABLENAME`, para definir o nome da tabela na qual o objeto será armazenado; `PRIMARYKEY`, para definir o nome do campo que será a chave primária; e `IDPOLICY` para definir como a chave primária será incrementada em novos registros. A constante `IDPOLICY` pode assumir os valores: `max` (incrementa a partir do último) e `serial` (não preenche o campo, utilizando o default programado para aquela tabela).

Criar um método construtor é opcional, mas importante, pois é nele que definimos os atributos do objeto. No método construtor, é importante executarmos o método construtor da classe-pai (`parent::__construct`). Além disso, ao executarmos o método `addAttribute()`, definiremos quais serão os atributos que a classe manipulará. Isto significa que somente atributos presentes no `addAttribute()` serão gravados, bem como retornados da base de dados. Quaisquer atributos fora desta lista serão ignorados. Assim, sempre que um atributo for acrescentado na tabela, deve ser também acrescentado nesta lista.

app/model/Customer.class.php

```
<?php
class Customer extends TRecord
{
    const TABLENAME = 'customer';
    const PRIMARYKEY= 'id';
    const IDPOLICY = 'max'; // {max, serial}

    public function __construct($id = NULL)
    {
        parent::__construct($id);

        parent::addAttribute('name');
        parent::addAttribute('address');
        parent::addAttribute('phone');
        parent::addAttribute('birthdate');
        parent::addAttribute('status');
        parent::addAttribute('email');
        parent::addAttribute('gender');
        parent::addAttribute('category_id');
        parent::addAttribute('city_id');
    }
}
```

3.3.3 Gravação de novo objeto

Para criar objetos a partir do momento em que temos a classe Active Record definida é muito simples. Para tal, basta criarmos um objeto da classe Active Record (`Customer` neste caso). A partir de então, basta indicarmos na forma de atributos do objeto, cada um dos campos da tabela que o objeto armazenará. Neste caso, o atributo `name` será utilizado para preencher a coluna de mesmo nome na tabela `customer`.

Ao final, o método `store()` é utilizado para armazenar o objeto na base de dados, gerando um `INSERT`. A partir desta gravação, o objeto recebe um `ID` em memória, correspondente à chave primária da base de dados. Neste caso, chamadas posteriores do método `store()` causariam um `UPDATE` sobre este mesmo objeto no banco de dados, pois o objeto já teria um `ID` em memória.

app/control/dbsamples/ObjectStore.class.php

```
<?php
class ObjectStore extends TPage
{
    public function __construct()
    {
        parent::__construct();

        try
        {
            // abre uma transação
            TTransaction::open('samples');

            // cria um novo objeto
            $giovani = new Customer;
            $giovani->name      = 'Giovanni Dall Oglia';
            $giovani->address   = 'Rua da Conceição';
            $giovani->phone     = '(51) 8111-2222';
            $giovani->birthdate = '2013-02-15';
            $giovani->status    = 'S';
            $giovani->email     = 'giovanni@dalloglio.net';
            $giovani->gender    = 'M';
            $giovani->category_id = '1';
            $giovani->city_id   = '1';
            $giovani->store(); // armazena o objeto

            new TMessage('info', 'Objeto armazenado com sucesso');
            TTransaction::close(); // fecha a transação.
        }
        catch (Exception $e)
        {
            new TMessage('error', $e->getMessage());
        }
    }
}
```

Além do método `store()`, também é possível armazenar um objeto por meio de um método estático simplificado chamado `create()`, disponível em cada uma das classes de modelo. No exemplo a seguir, repetindo a operação do exemplo anterior, armazenando um novo cliente na base de dados. O método `create()` recebe um vetor, onde cada um de seus índices representa o nome da coluna (atributo) do objeto. Este método não somente cria o objeto, mas também grava-o por meio do `store()`.

app/control/dbsamples/ObjectCreation.php

```
<?php
class ObjectCreation extends TPage
{
    public function __construct()
    {
        parent::__construct();
        try
        {
            TTransaction::open('samples'); // abre transação

            Customer::create( [ 'name'          => 'Antonio Dall Oglie',
                                'address'       => 'Rua da Conceicao',
                                'phone'         => '(51) 8111-2222',
                                'birthdate'     => '2013-02-15',
                                'status'        => 'S',
                                'email'         => 'antonio@dalloglio.net',
                                'gender'        => 'M',
                                'category_id'   => '1',
                                'city_id'       => '1' ] );

            new TMessage('info', 'Objeto armazenado com sucesso');
            TTransaction::close(); // Closes the transaction
        }
        catch (Exception $e)
        {
            new TMessage('error', $e->getMessage());
        }
    }
}
```

3.3.4 Carregamento de objeto

No exemplo anterior, vimos como transferir um novo objeto em memória para a base de dados. Neste exemplo, vamos fazer o caminho contrário, que é o de criar um objeto em memória a partir dos dados de um registro da base de dados. Neste caso, podemos identificar o ID (chave primária) do registro no momento de instanciarmos o objeto. Neste momento, automaticamente as colunas do registro na tabela são carregadas e disponibilizadas na forma de atributos do objeto em memória. Para demonstrar isto, no exemplo a seguir exibimos alguns atributos dos objetos em tela.

app/control/dbsamples/ObjectLoad.class.php

```
<?php
class ObjectLoad extends TPage
{
    public function __construct()
    {
        parent::__construct();
        try
        {
            TTransaction::open('samples'); // abre uma transação

            $customer = new Customer(4); // carrega o cliente 4
            echo 'Nome : ' . $customer->name . "<br>\n";
            echo 'Endereco : ' . $customer->address . "<br>\n";
            echo "<br>\n";

            $customer = new Customer(25); // carrega o cliente 25
            echo 'Nome : ' . $customer->name . "<br>\n";
            echo 'Endereco : ' . $customer->address . "<br>\n";
        }
    }
}
```

```
    TTransaction::close(); // fecha a transação.  
}  
catch (Exception $e)  
{  
    new TMessage('error', $e->getMessage());  
}  
}  
}
```

Ao instanciarmos um objeto, como fizemos no exemplo anterior (`new Customer`), presume-se que o objeto exista na base de dados. Caso o objeto não exista, uma exceção é lançada, e a execução segue para o bloco `catch`, onde esta exceção é tratada com a devida exibição da mensagem de erro em tela.

Porém, nem sempre queremos que uma exceção seja lançada quando um dado não for encontrado. Muitas vezes precisamos somente testar se o objeto existe ou não, para então tomar uma decisão.

O exemplo a seguir é uma variação do anterior, que permite testarmos a existência do objeto antes de sua exibição. Para tal, após abrirmos a transação, usamos o método `find()` da classe `Customer` (herdado de `TRecord`) para carregar o objeto de ID 4. Caso o objeto seja encontrado, o método `find()` retorna-o, caso contrário retorna `FALSE`. Assim, os dados somente serão exibidos se o objeto for encontrado. Caso não for encontrado, não teremos uma exceção.

[app/control/dbsamples/ObjectLoad2.class.php](#)

```

<?php
class ObjectLoad2 extends TPage
{
    public function __construct()
    {
        parent::__construct();
        try
        {
            TTransaction::open('samples');

            // carrega o cliente, retorna FALSE se não encontrar
            $customer = Customer::find(4);

            // verifica se o objeto foi encontrado
            if ($customer instanceof Customer)
            {
                echo 'Nome      : ' . $customer->name . "<br>\n";
                echo 'Endereço : ' . $customer->address . "<br>\n";
            }

            TTransaction::close();
        }
        catch (Exception $e)
        {
            new TMessage('error', $e->getMessage());
        }
    }
}

```

3.3.5 O padrão Lazy Load

Um objeto não vive isolado, ele é cercado por outros e ligado a estes por meio de relacionamentos como associações, composições, dentre outros. Quando carregamos um objeto em memória, podemos também carregar alguns objetos relacionados, mas devemos tomar alguns cuidados quanto a isso. Na maioria das vezes, carregar todos os objetos relacionados não é prudente, pois a partir de um único objeto, poderemos carregar indiretamente uma teia de objetos relacionados. Isto poderá influenciar o desempenho geral do sistema, tendo em vista vários usuários simultâneos.

Além disso, nem sempre precisamos acessar essa teia de objetos relacionados. Muitas vezes é também desnecessário carregar um conjunto de objetos e não fazer uso destes. Mas então como devemos proceder quando precisamos de uma informação associada que se encontra em outro objeto? Um Design Pattern para este caso é o Lazy initialization ou Lazy load, que consiste em atrasar a instânciação de um objeto até que ele seja realmente necessário, ou seja, até a primeira vez que ele seja acessado.

O Adianti Framework facilita a implementação do padrão Lazy Initialization e estimula o seu uso em relações de associação entre objetos. Neste exemplo, utilizaremos a relação de associação entre `Customer` (cliente) e `City` (cidade) para exemplificar.

Sempre que o desenvolvedor tem em memória o objeto `Customer` (cliente), ele poderá também acessar alguma propriedade do objeto `City` (cidade), mas não convém deixarmos este objeto pré-carregado em memória. Neste caso, instanciaremos o objeto cidade quando este for acessado pela primeira vez. Nas vezes subsequentes, o objeto já estará instaciado em memória. Para tal, basta criarmos métodos `getters`.

No Adianti Framework, todo método que inicia com “`get_`” é um método `getter` que simula um atributo. Assim, quando estivermos acessando o atributo `->city_name` do objeto cliente (desde que já não exista um atributo com o nome `city_name`), na verdade estaremos indiretamente chamando o método `get_city_name()`. E é exatamente neste momento, que o objeto `City` é instaciado pela primeira vez.

Neste caso, criaremos dois métodos: `get_city()`, para retornar o objeto cidade vinculado ao cliente e `get_city_name()`, para retornar o nome da cidade.

app/model/Customer.class.php

```
<?php
class Customer extends TRecord
{
    const TABLENAME = 'customer';
    const PRIMARYKEY= 'id';
    const IDPOLICY = 'max'; // {max, serial}

    private $city;
```

```

/**
 * Executado sempre que a propriedade ->city é acessada
 */
public function get_city()
{
    if (empty($this->city))
    {
        $this->city = new City($this->city_id);
    }

    return $this->city;
}

/**
 * Executado sempre que a propriedade ->city_name é acessada
 */
public function get_city_name()
{
    if (empty($this->city))
    {
        $this->city = new City($this->city_id);
    }

    return $this->city->name;
}

```

Agora criaremos um exemplo para demonstrar a utilização de um método para Lazy Load. Neste código, abrimos uma transação com a base de dados, e exibimos o atributo `->city->name`, que corresponde ao nome da cidade. Neste caso, o método `get_city()` é executado. Como é a primeira vez que o objeto `City` é necessário, ele é instanciado e carregado para a memória. Logo em seguida, o atributo `->city_name` é acessado e o método `get_city_name()` é executado. Como o objeto relacionado já se encontra instanciado em memória, ele não é carregado novamente e somente a informação vinculada (nome da cidade) é retornada.

app/control/dbsamples/ObjectLazy.class.php

```

<?php
class ObjectLazy extends TPage
{
    public function __construct()
    {
        parent::__construct();
        try {
            TTransaction::open('samples'); // abre uma transação

            $customer= new Customer(4); // carrega o cliente 4
            echo $customer->city->name; // chama get_city()

            echo '<br>';
            echo $customer->city_name; // chama get_city_name()

            TTransaction::close(); // fecha a transação.
        }
        catch (Exception $e)
        {
            new TMessage('error', $e->getMessage());
        }
    }
}

```

3.3.6 Alteração de objeto

Já vimos como salvar um novo objeto na base de dados, como carregar um registro do banco de dados na forma de objeto, e agora veremos como alterar um registro na base de dados. O primeiro passo é carregar o objeto em memória. Podemos utilizar duas formas para carregar um objeto: a primeira é por meio de um parâmetro no método construtor; a segunda é utilizando o método estático `find()`. A primeira abordagem lança uma exceção caso o objeto não for encontrado. A segunda abordagem apenas retorna `FALSE` quando não encontrar o objeto no banco de dados, o que é preferível neste caso por ser mais fácil de testar.

A partir do momento em que temos o objeto em memória (variável `$customer`), basta alterarmos os seus atributos diretamente, definindo novos valores. Após alterarmos os atributos, basta chamarmos o método `store()`. Mas como a classe `TRecord` sabe o que fazer com os dados? Como o objeto foi previamente carregado, ele possui um `ID` em memória. Neste caso, o framework sabe que deve proceder com uma alteração (`UPDATE`) e não uma criação (`INSERT`).

Obs: Note que todas as operações estão dentro de uma transação. Na medida em que uma operação falhar, todas as operações da transação são desfeitas.

app/control/dbsamples/ObjectUpdate.class.php

```
<?php
class ObjectUpdate extends TPage
{
    public function __construct()
    {
        parent::__construct();
        try
        {
            TTransaction::open('samples'); // abre uma transação

            // busca o cliente
            $customer = Customer::find(31);

            // se encontrou
            if ($customer) {
                $customer->phone = '51 8111-3333'; // muda o telefone
                $customer->store(); // armazena o objeto
            }

            new TMessage('info', 'Objeto atualizado');
            TTransaction::close(); // fecha a transação.
        }
        catch (Exception $e)
        {
            new TMessage('error', $e->getMessage());
        }
    }
}
```

3.3.7 Registro de log

Os frameworks automatizam várias tarefas comuns no desenvolvimento de aplicações, principalmente no acesso aos dados. Mesmo assim, em vários momentos precisamos compreender o que está se passando nos “bastidores”. Para permitir que possamos monitorar o que está acontecendo na camada de acesso aos dados, podemos ligar o registro de logs. O Framework vários tipos de logs: TXT, HTML, XML e STD, onde STD significa a saída padrão (em tela). Podemos facilmente estender o framework e criar outros tipos de registro. Para mostrar o registro de logs, vamos inserir uma nova cidade no banco de dados. Primeiramente, criamos o Active Record para manipulação de `City` (cidades).

`app/model/City.class.php`

```
<?php
class City extends TRecord
{
    const TABLENAME = 'city';
    const PRIMARYKEY = 'id';
    const IDPOLICY = 'max';
}
```

Neste exemplo, abrimos uma transação com a base de dados, definimos o tipo de log (`TLoggerTXT` para arquivo TXT), o caminho do arquivo de log, e em seguida realizamos a persistência do objeto recém-criado por meio do método `store()`. A qualquer momento podemos escrever manualmente no log por meio do método `log()`.

`app/control/dbsamples/RegisterLog.class.php`

```
<?php
class RegisterLog extends TPage
{
    public function __construct()
    {
        parent::__construct();
        try
        {
            TTransaction::open('samples'); // abre uma transação

            // define o log
            TTransaction::setLogger(new TLoggerTXT('/tmp/log.txt'));
            TTransaction::log("** inserindo cidade");

            $cidade = new City; // cria novo objeto
            $cidade->name = 'Porto Alegre';
            $cidade->state_id = '1';
            $cidade->store(); // armazena o objeto

            new TMessage('info', 'Objeto armazenado com sucesso');
            TTransaction::close(); // fecha a transação.
        }
        catch (Exception $e)
        {
            new TMessage('error', $e->getMessage());
        }
    }
}
```

A seguir, podemos conferir o que foi registrado no arquivo de log.

/tmp/log.txt

```
2012-07-12 17:06:42 :: ** inserindo cidade
2012-07-12 17:06:42 :: SELECT max(id) as id FROM city
2012-07-12 17:06:42 :: INSERT INTO city (name, id) values ('Porto Alegre', 2)
```

Como vimos no exemplo anterior, é bastante simples efetuar o registro de logs dos comandos que são executados dentro de uma transação. As classes de log coletam os comandos SQL gerados pelo framework e direcionam esses logs para uma saída. A classe **TLoggerTXT** direciona a saída para um arquivo TXT, e a classe **TLoggerSTD** para log em tela. Outra maneira de registrar os logs é por meio de uma função anônima. Uma função anônima é uma função sem nome próprio e que é definida em tempo de execução, podendo ser passada como parâmetro para outros métodos.

No exemplo a seguir, após abrirmos transação com a base de dados, definimos uma função anônima como parâmetro do método **setLoggerFunction()**. A função anônima recebe como parâmetro uma mensagem, que por sua vez representa um comando SQL gerado pelo framework. A função anônima não é chamada nesse instante, mas é armazenada para execução posterior, sempre que uma operação com o banco de dados (ex: **store**) for realizada. Ao executar este exemplo, você poderá ver os logs em tela, já que a função usa o comando **echo**.

app/control/dbsamples/RegisterLogFunction.class.php

```
<?php
class RegisterLogFunction extends TPage
{
    public function __construct()
    {
        parent::__construct();
        try
        {
            TTransaction::open('samples'); // abre transação

            // define uma função de log
            TTransaction::setLoggerFunction( function($message) {
                echo $message . '<br>';
            });

            TTransaction::log("inserindo cidade...");

            // declara e armazena o objeto
            $cidade = new City;
            $cidade->name = 'Porto Alegre';
            $cidade->store();

            TTransaction::close(); // fecha transação
        }
        catch (Exception $e)
        {
            new TMessage('error', $e->getMessage());
        }
    }
}
```

Obs: A finalidade do registro de logs é fundamentalmente debug. Se você deseja armazenar os logs do sistema para auditoria, veja que o Template abordados ao final do livro já possuem essa funcionalidade implementada, bastando configurar e utilizar.

O registro de log também pode ser feito pelo método `dump()`. Este método habilita o dump dos comandos da transação diretamente em tela. Se um caminho de arquivo for informado como parâmetro, então os comandos serão armazenados dentro do arquivo.

app/control/dbsamples/RegisterLogDump.class.php

```
<?php
class RegisterLogDump extends TPage
{
    public function __construct()
    {
        parent::__construct();
        try {
            TTransaction::open('samples'); // abre uma transação
            // screen dump (uncomment to save file)
            TTransaction::dump( /* '/tmp/log.txt' */ );

            $cidade = new City; // cria novo objeto
            $cidade->name = 'Porto Alegre';
            $cidade->state_id = '1';
            $cidade->store(); // armazena o objeto

            new TMessage('info', 'Objeto armazenado com sucesso');
            TTransaction::close(); // fecha a transação.
        }
        catch (Exception $e) {
            new TMessage('error', $e->getMessage());
        }
    }
}
```

3.3.8 Encapsulamento

Encapsulamento é uma técnica que visa construir objetos que saibam se “proteger” do mundo externo, ou seja, se comportem como cápsulas. Mas quem é o mundo externo? Programadores que usarão estes objetos em outras rotinas. Você deve ter percebido que no Adianti Framework a forma utilizada para definir as propriedades de um objeto Active Record é por meio da atribuição direta como em: `$cidade->name = 'Porto Alegre'`. A atribuição direta nos confere agilidade no desenvolvimento, mas nem sempre podemos atribuir valores aos atributos de um objeto indiscriminadamente, pois uma validação se faz necessária, como na atribuição de datas.

No Framework, o encapsulamento é realizado por meio de métodos *setters* iniciados por `“set_”`. No exemplo a seguir, declaramos um método chamado `set_birthdate()`. Isto significa que este método será executado sempre que atribuirmos algum valor ao atributo `->birthdate`. O método poderá decidir se a atribuição irá ou não ocorrer. Caso ocorra, basta atribuirmos ao vetor `->data`, que é um vetor interno de dados do Active Record que armazena seus atributos e é indexado pelo nome do atributo. Quando a atribuição não é válida lançamos uma exceção que será tratada pelo controlador.

[app/model/Customer.class.php](#)

```
<?php
/**
 * Active Record para Customer (clientes)
 */
class Customer extends TRecord
{
    // ... métodos já desenvolvidos ...

    public function set_birthdate($value)
    {
        $parts = explode('-', $value);
        if (checkdate($parts[1], $parts[2], $parts[0]))
        {
            $this->data['birthdate'] = $value;
        }
        else
        {
            throw new Exception("Não pode atribuir '{$value}' em birthdate");
        }
    }
}
```

Obs: Neste e em outros exemplos do livro, não repetiremos códigos já desenvolvidos, apresentando somente novos trechos. Veja a frase “métodos já desenvolvidos”. Os exemplos completos encontram-se para download na aplicação Tutor.

Este exemplo demonstra a utilização do encapsulamento. Após a abertura da transação com a base de dados, atribuímos o nome do objeto e a data de nascimento. Neste momento, o método `set_birthdate()` é executado e uma exceção é lançada, uma vez que a atribuição é inválida, gerando uma mensagem de erro (`new TMessage`).

[app/control/dbsamples/ObjectEncapsulation.class.php](#)

```
<?php
class ObjectEncapsulation extends TPage
{
    public function __construct()
    {
        parent::__construct();
        try
        {
            // abre uma transação
            TTransaction::open('samples');

            $giovani = new Customer;
            $giovani->name      = 'Giovanni Dall Oglie';
            $giovani->birthdate = '2013-32-40';

            // mensagem de sucesso
            new TMessage('info', 'Objeto armazenado com sucesso');

            // fecha a transação.
            TTransaction::close();
        }
        catch (Exception $e) {
            new TMessage('error', $e->getMessage());
        }
    }
}
```

3.3.9 Exclusão de objeto

Neste próximo exemplo, vamos demonstrar como excluir um determinado registro da base de dados. Para tal, existem duas formas possíveis. Na primeira forma, primeiramente carregamos o objeto em memória, identificando seu ID no método construtor e em seguida chamamos o método `delete()`. Se já temos o objeto em memória, esta alternativa é interessante. Mas se o objetivo é somente excluir o objeto, não é interessante ter de carregá-lo antes somente para a exclusão.

A segunda forma de excluir um registro é instanciando um objeto vazio (`new Customer`) e em seguida chamar o método `delete()` identificando o ID do registro como parâmetro. Neste caso, o Framework não carrega o registro em memória, mas exclui o mesmo diretamente.

[app/control/dbsamples/ObjectDelete.class.php](#)

```
<?php
class ObjectDelete extends TPage
{
    public function __construct()
    {
        parent::__construct();
        try
        {
            // abre uma transação
            TTransaction::open('samples');

            $customer = new Customer(40); // carrega o objeto
            $customer->delete(); // exclui o objeto já carregado

            $customer = new Customer;
            $customer->delete(41); // exclui o objeto sem carregar

            new TMessage('info', 'Objeto excluído');
            TTransaction::close(); // fecha a transação.
        }
        catch (Exception $e)
        {
            new TMessage('error', $e->getMessage());
        }
    }
}
```

Obs: Poderíamos também utilizar o método `find()` para efetuar a carga do objeto.

3.3.10 Primeiros e últimos Identificadores

O Adianti Framework, por meio de sua classe `TRecord`, fornece um mecanismo para gravação de objetos que dispensa termos de escrever comandos SQL com Inserts, Updates ou Deletes na mão. A classe `TRecord` também trata, dentre outras questões, da geração de novos ID's para um novo registro, por meio da estratégia `MAX` (último +1) ou `SERIAL` (sequência do banco de dados).

Apesar de já termos um bom nível de automatização, por vezes é interessante termos alguns recursos de baixo nível disponíveis. Um recurso não tão usual que a classe **TRecord** disponibiliza é a descoberta do primeiro e últimos objetos de uma classe no banco de dados, que pode ser realizada pelos métodos **first()** e **last()**. Estes métodos retornam respectivamente o primeiro e último objeto, baseado no menor e maior valor da chave primária da tabela.

No exemplo a seguir, é aberta uma transação com a base de dados e em seguida é utilizado o método **first()** para retornar o primeiro objeto encontrado (menor valor de chave primária), e alguns atributos (**id**, **e name**) são exibidos. Em seguida, é utilizado o método **last()** para retornar o último objeto encontrado (maior valor de chave primária), e também alguns de seus atributos (**id**, **name**) são exibidos. O atributo de chave primária é definido pela constante **PRIMARYKEY** da classe **Customer**.

app/control/dbsamples/ObjectFirstLast.class.php

```
<?php
class ObjectFirstLast extends TPage
{
    public function __construct()
    {
        parent::__construct();
        try
        {
            TTransaction::open('samples'); // abre transação

            $first = Customer::first();
            print '<b>First ID</b>: ' . $first->id . '<br>';
            print '<b>First Name</b>: ' . $first->name . '<br>';

            echo '<br>';
            $last = Customer::last();
            print '<b>Last ID</b>: ' . $last->id . '<br>';
            print '<b>Last Name</b>: ' . $last->name . '<br>';

            TTransaction::close(); // closes transaction
        }
        catch (Exception $e)
        {
            new TMessage('error', $e->getMessage());
        }
    }
}
```

3.3.11 Conversão entre Active Record e Array

Quando falamos de persistência de objetos, normalmente estamos nos referindo às bases de dados relacionais, mas estes não são os únicos meios de armazenamento de objetos. Podemos armazenar objetos em formatos como **XML**, **JSON**, dentre outros. Para facilitar a conversão de objetos entre formatos, a classe **TRecord** permite o intercâmbio entre o objeto e o formato de array. O formato de array pode ser posteriormente convertido em outro formato de maneira muito simples.

A classe `TRecord` fornece dois métodos básicos para conversão de objetos que são: `toArray()`, que retorna um vetor contendo os atributos do objeto; e `fromArray()` que carrega os atributos internos de um objeto a partir de um vetor. Em ambos os casos, os índices do vetor devem corresponder aos atributos do objeto manipulado.

No exemplo a seguir, é aberta uma transação com a base de dados e carregado um objeto (4). Este objeto deve existir neste momento, caso contrário uma exceção será lançada. Em seguida é exibindo em tela o resultado da conversão do objeto em vetor. Logo após, é declarado um vetor contendo alguns dados de um cliente fictício. Veja que os índices do vetor correspondem aos atributos do objeto. Por fim, estamos declarando um objeto (`$customer2`) e alimentando seus atributos com os valores deste vetor, por meio do método `fromArray()`. Os dados somente são persistidos após o fechamento da transação (`Ttransaction::close()`).

app/control/dbsamples/ObjectFromArray.class.php

```
<?php
class ObjectFromArray extends TPage
{
    public function __construct()
    {
        parent::__construct();
        try {
            TTransaction::open('samples'); // abre a transação

            // Converte um Active Record em um array
            $customer = new Customer(4);
            print_r($customer->toArray());

            // Usa um array para preencher o Active Record
            $test = array();
            $test['name'] = 'Customer from array';
            $test['address'] = 'Address from array';
            $test['category_id'] = 1;
            $test['city_id'] = 1;
            $test['birthdate'] = date('Y-m-d');

            $customer2 = new Customer();
            $customer2->fromArray($test);
            $customer2->store();
            TTransaction::close(); // fecha a transação
        }
        catch (Exception $e) {
            new TMessage('error', $e->getMessage());
        }
    }
}
```

3.3.12 Conversão para JSON

No exemplo anterior, vimos como converter um objeto para Array, e também como preenchê-lo a partir de um Array. Outro formato cada vez mais popular é o JSON (*JavaScript Object Notation*), utilizado muito na transferência de dados entre aplicações e entre o back-end e front-end da mesma aplicação.

Neste exemplo veremos como é fácil converter um objeto para o formato JSON. Após a abertura da transação com a base de dados, o objeto é carregado por meio do método estático `find()`. Caso seja encontrado, usamos simplesmente um `print_r()` para exibir o conteúdo exibido é o resultado do método `toJson()`, que converte os atributos do objeto para o formato JSON.

`app/control/dbsamples/ObjectToJson.class.php`

```
<?php
class ObjectToJson extends TPage
{
    public function __construct()
    {
        parent::__construct();
        try
        {
            TTransaction::open('samples');

            // Carrega o objeto
            $customer = Customer::find(4);

            // verifica se foi encontrado
            if ($customer)
            {
                print_r( $customer->toJson() ); // exibe no formato JSON
            }

            TTransaction::close();
        }
        catch (Exception $e) {
            new TMessage('error', $e->getMessage());
        }
    }
}
```

3.3.13 Hook methods

Como vimos anteriormente, os métodos do framework automatizam várias operações como o carregamento (`find`) e gravação (`store`) de objetos em base de dados. Essa automatização nos dá grande agilidade e velocidade, uma vez que não precisamos mais nos preocupar em escrever instruções SQL para tal. Por outro lado, essa automatização torna a persistência rígida, tirando a possibilidade de realizar pequenas modificações nos métodos padronizados fornecidos pelo framework (Ex: `store()`, `find()`). Em algumas situações, é importante “personalizarmos” estas ações, adicionando comportamentos específicos.

Um exemplo que pode ser dado é o registro automático de logs de alterações de registros, que será abordado no Template. O log automático de alteração de registros deve ser realizado sempre que um objeto é alterado (`store`), independente de qual parte do sistema esse método for executado. Assim, é importante podermos “adicionar” esse comportamento ao método `store()`, por exemplo.

Para adicionarmos comportamentos “específicos” a métodos padronizados, o framework disponibiliza *Hook Methods*. Um *Hook method* (método de gancho) é chamado automaticamente a partir de um método padrão do framework. Um *Hook Method* funciona como um evento que fica “observando” o método padrão e permite adicionar um comportamento a um ponto do processo (Ex: salvar, ou excluir registro).

O framework disponibiliza alguns *Hook Methods*, que são: `onBeforeLoad()`, que é executado antes de um registro ser carregado; `onAfterLoad()`, executado após o registro ser carregado; `onBeforeStore()`, executado antes de um registro ser salvo; `onAfterStore()`, executado após o registro ser salvo; `onBeforeDelete()`, executado antes de um registro ser excluído; e `onAfterDelete()`, executado após um registro ser excluído.

Exemplo de Active Record

```
<?php
class SuaClasseActiveRecord extends TRecord
{
    const TABLENAME = 'tablename';
    const PRIMARYKEY= 'id';
    const IDPOLICY = 'max'; // {max, serial}

    public function __construct($id = NULL)
    {
        parent::__construct($id);
        parent::addAttribute('name');
    }

    public function onBeforeLoad($id)
    {
        file_put_contents('/tmp/log.txt', "onBeforeLoad: $id\n", FILE_APPEND);
    }

    public function onAfterLoad($object)
    {
        file_put_contents('/tmp/log.txt', 'onAfterLoad:' .
            json_encode($object)."\\n", FILE_APPEND);
    }

    public function onBeforeStore($object)
    {
        file_put_contents('/tmp/log.txt', 'onBeforeStore:' .
            json_encode($object)."\\n", FILE_APPEND);
    }

    public function onAfterStore($object)
    {
        file_put_contents('/tmp/log.txt', 'onAfterStore:' .
            json_encode($object)."\\n", FILE_APPEND);
    }

    public function onBeforeDelete($object)
    {
        file_put_contents('/tmp/log.txt', 'onBeforeDelete:' .
            json_encode($object)."\\n", FILE_APPEND);
    }

    public function onAfterDelete($object)
    {
        file_put_contents('/tmp/log.txt', 'onAfterDelete:' .
            json_encode($object)."\\n", FILE_APPEND);
    }
}
```

Obs: Dentro de um *Hook Method*, você pode escrever qualquer funcionalidade. No exemplo a seguir, estamos somente registrando as ocorrências dos eventos em um arquivo-texto. Você poderá acrescentar *Hook Methods* em qualquer subclasse de `Trecord`.

3.3.14 Renderização e cálculos

Neste exemplo, serão demonstrados dois métodos disponibilizados para todos objetos do tipo `TRecord`, que são `render()` e `evaluate()`. O método `render()` recebe uma máscara de exibição (string) que pode conter atributos do objeto entre chaves, no formato de um template, e retorna a string resultante, preenchida com os atributos do objeto. Já o método `evaluate()` recebe uma fórmula iniciando por igual “=”, que pode conter atributos numéricos entre chaves, e retorna o resultado deste cálculo. Com isso, podemos executar o resultado de uma fórmula cadastrada pelo usuário em uma área de preferências, por exemplo.

[app/control/dbsamples/ObjectRender.php](#)

```
<?php
class ObjectRender extends TPage
{
    public function __construct()
    {
        parent::__construct();
        try
        {
            TTransaction::open('samples');

            $product = Product::find(4);

            // renderiza atributos entre chaves
            echo $product->render('The product <b>{id}</b> is <u>{description}</u>');

            echo '  
';

            // avalia a fórmula, substitui atributos entre chaves
            echo $product->evaluate('= {sale_price} * {stock}');

            TTransaction::close();
        }
        catch (Exception $e)
        {
            new TMessage('error', $e->getMessage());
        }
    }
}
```

3.3.15 CreatedAt e UpdatedAt

As constantes `CREATEDAT` e `UPDATEDAT` são opcionais. Quando utilizadas, elas definem o nome dos campos que armazenam respectivamente a data/hora de criação do registro, bem como a data/hora de sua última alteração. Estas informações são registradas e atualizadas automaticamente pelo framework quando o registro for criado ou alterado.

app/model/Customer.class.php

```
<?php
class Customer extends TRecord
{
    const TABLENAME = 'customer';
    const PRIMARYKEY = 'id';
    const IDPOLICY = 'max'; // {max, serial}

    const CREATEDAT = 'created_at';
    const UPDATEDAT = 'updated_at';
}
```

3.4 Manipulação de coleções

3.4.1 O padrão Repository

O padrão Active Record nos permite trabalhar com um objeto que representa uma linha do banco de dados, ou seja, ele individualiza os registros fazendo com que cada um seja representado por um objeto em memória. E quando desejamos trabalhar com uma coleção (um conjunto) de objetos? Podemos utilizar o padrão Repository. O padrão Repository é implementado por uma classe que disponibiliza uma interface para manipular coleções de objetos como representado pela figura a seguir.

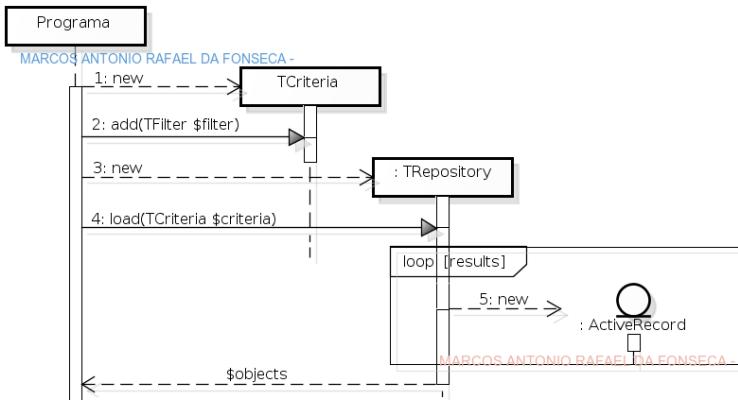


Figura 9 Carregamento de objetos com repositório

Como podemos ver na figura, a aplicação (programa) interage com uma classe que implementa o padrão Repository (**TRepository**). Esta classe disponibiliza métodos para carregar (**load**), contar (**count**) e excluir (**delete**) coleções de objetos. Antes de realizar a operação desejada, é necessário identificar o conjunto de objetos a ser manipulado. A identificação do conjunto de objetos é realizada por um critério (objeto **TCriteria**). Um critério identifica regras de seleção de objetos na base de dados.

A classe **TRepository** realiza um acesso à base de dados e interage com os objetos selecionados pelo critério. No exemplo demonstrado pela figura estamos carregando objetos (**load**). Neste caso, vários objetos do tipo Active Record são instanciados (conforme o critério) e retornados para a aplicação como vetor.

3.4.2 API de critérios

Como visto, ao trabalharmos com um repositório, é importante dominarmos a definição de critérios para seleção de objetos. A seguir, são demonstrados vários exemplos de critérios de seleção de objetos. A classe **TCriteria** possui o método **add()**, que permite adicionar regras de filtragem ao critério. A classe **TFilter** identifica um filtro único utilizado para criar regras lógicas. O método **add()** adiciona por padrão vários filtros, utilizando o operador **AND**, a menos que identifiquemos apóis o parâmetro do filtro, o operador a ser utilizado.

O primeiro exemplo demonstra a criação de um critério com o operador **OR**, onde se compara a idade (**age**). O segundo exemplo demonstra a utilização da cláusula **BETWEEN**, que exige um parâmetro a mais. O terceiro exemplo demonstra um critério com regras envolvendo vetores numéricos e operadores como **IN** e **NOT IN**. O quarto exemplo demonstra um critério com comparação de textos com o operador **like**. O quinto exemplo demonstra o operador **IS NOT NULL**. O sexto exemplo demonstra a utilização de critérios com vetores de strings. O sétimo exemplo demonstra a utilização de **subqueries** na composição de filtros, neste caso não são utilizadas aspas para formação do parâmetro. O oitavo exemplo demonstra como forçar a não utilização de aspas (escape) na formação da expressão, por meio da utilização do prefixo **“NOESC:**” no início do terceiro parâmetro. Por fim, o nono exemplo demonstra a utilização de critérios compostos, ou seja, a constituição de um objeto de critério (**TCriteria**) utilizando operadores lógicos (**AND**, **OR**), a partir de outros critérios já definidos.

```
app/control/dbsamples/ObjectCriteria.class.php


---


$criteria = new TCriteria;                                     # exemplo 1
$criteria->add(new TFilter('age', '<', 16), TExpression::OR_OPERATOR);
$criteria->add(new TFilter('age', '>', 60), TExpression::OR_OPERATOR);
echo $criteria->dump(); // (age < 16 OR age > 60)

$criteria = new TCriteria;                                     # exemplo 2
$criteria->add(new TFilter('age', 'BETWEEN', 16, 60));
echo $criteria->dump(); // (age BETWEEN 16 AND 60)

$criteria = new TCriteria;                                     # exemplo 3
$criteria->add(new TFilter('age','IN', array(24,25,26)));
$criteria->add(new TFilter('age','NOT IN', array(10)));
echo $criteria->dump(); // (age IN (24,25,26) AND age NOT IN (10))

$criteria = new TCriteria;                                     # exemplo 4
$criteria->add(new TFilter('nome', 'like', 'pedro%'), TExpression::OR_OPERATOR);
$criteria->add(new TFilter('nome', 'like', 'maria%'), TExpression::OR_OPERATOR);
echo $criteria->dump(); // (nome like 'pedro%' OR nome like 'maria%')

$criteria = new TCriteria;                                     # exemplo 5
$criteria->add(new TFilter('cellphone', 'IS NOT', NULL));
$criteria->add(new TFilter('gender', '=', 'F'));
echo $criteria->dump(); // (cellphone IS NOT NULL AND gender = 'F')

$criteria = new TCriteria;                                     # exemplo 6
$criteria->add(new TFilter('state', 'IN', array('RS', 'SC', 'PR')));
$criteria->add(new TFilter('state', 'NOT IN', array('AC', 'PI')));
echo $criteria->dump(); // (state IN ('RS','SC','PR') AND state NOT IN ('AC','PI'))
```

```

$criteria = new TCriteria;                                # exemplo 7
$criteria->add(new TFilter('id', 'IN', '(SELECT customer_id FROM purchases')));
echo $criteria->dump(); // (id IN (SELECT customer_id FROM purchases))

$criteria = new TCriteria;                                # exemplo 8
$criteria->add(new TFilter('birthdate', '<=', "NOESC:date(now()) - '2
years':interval"));
echo $criteria->dump(); // (birthdate <= date(now()) - '2 years':interval)

$criteria1 = new TCriteria;                               # exemplo 9
$criteria1->add(new TFilter('gender', '=', 'F'));
$criteria1->add(new TFilter('age', '>', '18'));

$criteria2 = new TCriteria;
$criteria2->add(new TFilter('gender', '=', 'M'));
$criteria2->add(new TFilter('age', '<', '16'));

$criteria = new TCriteria;
$criteria->add($criteria1, TExpression::OR_OPERATOR);
$criteria->add($criteria2, TExpression::OR_OPERATOR);
echo $criteria->dump(); // ((gender='F' AND age>'18') OR (gender='M' AND age<'16'))

```

Obs: Apesar de ser utilizado neste exemplo, o método `dump()` raramente será utilizado na prática. Este é um método executado internamente pela classe `TRepository`.

Existe uma maneira alternativa de criação de critérios pelo método `create()`. Este método recebe como parâmetro um vetor, que contém filtros simples (todos comparados com igual “`=`”). Este método pode ser utilizado quando a busca somente utilizar operador de comparação igual “`=`” e operador lógico `AND` entre os operandos.

Critério rápido

```

$criteria = TCriteria::create( [ 'state_id' => '5',
                                'gender'    => 'M' ] );
// exibe o resultado do critério
var_dump($criteria->dump());
// resultado: "(state_id = '5' AND gender = 'M')"

```

3.4.3 Contagem de coleções

Em diversas situações precisamos contar quantos objetos satisfazem um determinado critério de seleção. Para isso, pode ser utilizado o método `count()` da classe `TRepository`. Neste exemplo, contaremos quantos registros do Active Record `Customer` possuem o nome iniciando por “Rafael” ou “Ana”. Para realizar esse filtro, criamos um objeto para constituir um critério (`TCriteria`), e adicionamos dois filtros (`TFilter`) por meio do método `add()`. Após definirmos nosso critério de seleção de objetos, criamos um objeto para gerenciar a coleção de objetos do tipo `Customer`, instanciando o objeto `TRepository`. A partir desse ponto, o objeto sabe qual o Active Record que manipulará.

O objetivo desse exemplo é contar quantos objetos satisfazem um critério. Para tal, usamos o método `count()` da classe `TCriteria`. Este método recebe como parâmetro um critério (objeto `TCriteria`) e verifica na base de dados a quantidade de objetos que passam pelo critério de filtragem. Logo após, a quantidade é exibida em tela.

Obs: O método `count()` apenas conta os registros, não carregando-os em memória.

app/control/dbsamples/CollectionCount.class.php

```
<?php
class CollectionCount extends TPage
{
    public function __construct()
    {
        parent::__construct();
        try {
            TTransaction::open('samples'); // abre uma transação
            $criteria = new TCriteria;
            $criteria->add(new TFilter('name', 'like', 'Rafael%'),
                           TExpression::OR_OPERATOR);
            $criteria->add(new TFilter('name', 'like', 'Ana%'),
                           TExpression::OR_OPERATOR);

            $repository = new TRepository('Customer');
            $count = $repository->count($criteria);

            new TMessage('info', "Total de clientes encontrados: {$count} <br>\n");
            TTransaction::close(); // fecha a transação.
        }
        catch (Exception $e) {
            new TMessage('error', $e->getMessage());
        }
    }
}
```

O exemplo a seguir apresenta uma forma alternativa ainda mais simplificada para realizar a contagem de objetos. A partir do método estático `where()` da classe `TRecord`, podemos estabelecer uma cadeia de instruções `where()`, onde cada cláusula é adicionada ao critério por um operador de `AND` (a não ser que o quarto parâmetro, que é opcional, seja informado). Ao fim da cadeia de `where()`, podemos executar o método `count()`, retornando a quantidade de registros encontrados pelo filtro.

app/control/dbsamples/CollectionStaticSimpleCount.class.php

```
<?php
class CollectionStaticSimpleCount extends TPage
{
    public function __construct()
    {
        parent::__construct();
        try {
            TTransaction::open('samples');

            $count = Customer::where('gender', '=', 'M')
                      ->where('name', 'like', 'A%')
                      ->count();

            print( $count );
            TTransaction::close();
        }
        catch (Exception $e) {
            new TMessage('error', $e->getMessage());
        }
    }
}
```

3.4.4 Carregamento de coleções

Nesse exemplo, veremos como carregar objetos em memória. A partir do momento em que temos uma coleção de objetos em memória, podemos realizar uma série de operações sobre estes, afinal são objetos Active Record. Neste exemplo, carregaremos em memória todos objetos `Customer`, cujo gênero é feminino. Para tal, após a abertura da transação com a base de dados, criaremos o critério de seleção de registros (`TCriteria`) e adicionaremos um filtro (`gender='F'`). Após definirmos o critério de seleção, instanciaremos o repositório (`TRepository`) para manipular a classe `Customer`. Logo em seguida, executaremos o método `load()`, cuja finalidade é carregar em memória um vetor com todos os objetos que satisfazem o critério determinado. Após, percorreremos os objetos retornados, e exibiremos o código (`id`) e nome (`name`).

`app/control/dbsamples/CollectionLoad.class.php`

```
<?php
class CollectionLoad extends TPage
{
    public function __construct()
    {
        parent::__construct();
        try {
            TTransaction::open('samples'); // abre uma transação

            $criteria = new TCriteria;
            $criteria->add(new TFilter('gender', '=', 'F'));

            $repository = new TRepository('Customer');
            $customers = $repository->load($criteria);

            foreach ($customers as $customer)
            {
                echo $customer->id . ' - ' . $customer->name . '<br>';
            }
            TTransaction::close(); // fecha a transação.
        }
        catch (Exception $e) {
            new TMessage('error', $e->getMessage());
        }
    }
}
```

O framework também permite o carregamento de objetos por um atalho, que é o método `getObjects()`, podendo ser chamado de maneira estática a partir de qualquer Active Record, recebendo o critério e devolvendo o `array` de objetos.

```
<?php
$customers = Customer::getObjects($criteria);
```

O exemplo a seguir apresenta uma forma alternativa ainda mais simplificada para realizar o carregamento de objetos. A partir do método estático `where()` da classe `TRecord`, podemos estabelecer uma cadeia de instruções `where()`, onde cada cláusula é adicionada ao critério por um operador de `AND` (a não ser que o quarto parâmetro, que é opcional, seja informado). Ao fim da cadeia de `where()`, podemos executar o método `load()`, retornando um vetor contendo os registros encontrados pelo filtro.

app/control/dbsamples/CollectionStaticSimpleLoad.class.php

```
<?php
class CollectionStaticSimpleLoad extends TPage
{
    public function __construct()
    {
        parent::__construct();
        try {
            TTransaction::open('samples');

            // carrega clientes
            $customers = Customer::where('gender', '=', 'M')
                ->where('name', 'like', 'A%')
                ->load();

            foreach ($customers as $customer)
            {
                echo $customer->id . ' - ' . $customer->name . '<br>';
            }
            TTransaction::close(); // fecha a transação.
        }
        catch (Exception $e)
        {
            new TMessage('error', $e->getMessage());
        }
    }
}
```

Existe ainda uma forma alternativa de carregamento de objetos que se utiliza do método `select()` encadeado anteriormente ao `where()`. O método `select()` permite determinar exatamente quais colunas serão carregadas em memória, reduzindo portanto o volume de memória necessário para realizar algumas operações, uma vez que carregará menos atributos do banco de dados.

app/control/dbsamples/CollectionStaticSimpleLoad.class.php

```
<?php
class CollectionStaticSimpleLoad extends TPage
{
    public function __construct()
    {
        parent::__construct();
        try {
            TTransaction::open('samples');

            // carrega clientes
            $customers = Customer::select('id', 'name')
                ->where('gender', '=', 'M')
                ->where('name', 'like', 'A%')
                ->load();

            foreach ($customers as $customer)
            {
                echo $customer->id . ' - ' . $customer->name . '<br>';
            }
            TTransaction::close(); // fecha a transação.
        }
        catch (Exception $e)
        {
            new TMessage('error', $e->getMessage());
        }
    }
}
```

3.4.5 Carregamento paginado e ordenado

No exemplo anterior, vimos como carregar objetos usando filtragem por meio das classes `TCriteria` e `TFilter`. Para isso, usamos o método `add()` da classe `TCriteria`, que permite adicionar um ou vários objetos da classe `TFilter`. Além de permitir adicionar filtros por meio do método `add()`, podemos alterar as propriedades de um critério de seleção de dados.

Neste exemplo, estamos novamente carregando objetos da classe `Customer`, mas estamos definindo as propriedades do carregamento por meio do método `setProperty()` da classe `TCriteria`, que são: `limit` (define o máximo de registros a serem retornados na consulta), `offset` (define a quantidade inicial de registros a serem ignorados), e `order` (define o campo da ordenação). Desta forma, podemos realizar buscas paginadas, onde somente uma faixa de dados é buscada. Neste exemplo, serão apresentados todos os registros do 21 ao 30.

`app/control/dbsamples/CollectionLoadProperty.class.php`

```
<?php
class CollectionLoadProperty extends TPage
{
    public function __construct()
    {
        parent::__construct();
        try
        {
            // abre uma transação
            TTransaction::open('samples');

            $criteria = new TCriteria;
            $criteria->setProperty('limit' , 10);
            $criteria->setProperty('offset', 20);
            $criteria->setProperty('order' , 'id');

            $repository = new TRepository('Customer');
            $customers = $repository->load($criteria);

            foreach ($customers as $customer)
            {
                echo $customer->id . ' - ' . $customer->name . '<br>';
            }
            TTransaction::close(); // fecha a transação.
        }
        catch (Exception $e)
        {
            new TMessage('error' , $e->getMessage());
        }
    }
}
```

O próximo exemplo demonstra de uma maneira ainda mais fácil como realizar a mesma operação (carga paginada de objetos). Neste exemplo, após a abertura da transação, usamos o método `orderBy()` de `TRecord` para definirmos a ordenação da busca, o método `take()` para definir o `limit`, o método `skip()` para definir o `offset`, bem como ao final o método `load()` para efetuar a carga do Array de objetos.

Obs: Para realizar um filtro na busca, basta iniciarmos a operação pelo método `where()`, e em seguida utilizarmos os demais (`take`, `skip`, `orderBy`, etc).

app/control/dbsamples/CollectionSimpleLoadProperty.class.php

```
<?php
class CollectionSimpleLoadProperty extends TPage
{
    public function __construct()
    {
        parent::__construct();
        try
        {
            TTransaction::open('samples'); // abre uma transação

            // carrega ordenado pelo id, com limit 10 e offset 20
            $customers = Customer::orderBy('id')->take(10)->skip(20)->load();

            if ($customers)
            {
                foreach ($customers as $customer)
                {
                    echo $customer->id . ' - ' . $customer->name . '<br>';
                }
            }

            TTransaction::close(); // fecha a transação
        }
        catch (Exception $e)
        {
            new TMessage('error', $e->getMessage());
        }
    }
}
```

3.4.6 Carregamento de listas simples de dados

Até o momento, sempre que falamos em coleções, falamos em vetores de objetos. Mas as vezes, tudo que nossa aplicação precisa é de um vetor simples, contendo código e nome de clientes, ou de produtos. Para fornecer listas simples de dados, as classes `TRecord` e `TRepository` fornecem um método chamado `getIndexedArray()`. Este método retorna um vetor indexado, sendo que podemos escolher qual atributo será utilizado como índice, e qual atributo será utilizado como valor deste vetor.

No exemplo a seguir, temos quatro formas diferentes de utilização deste método. Na primeira, temos a obtenção de uma lista de todos os produtos, em um vetor cujo índice será o código do produto. Na segunda forma, demonstramos a possibilidade de utilizar máscaras tanto na formação da string de chave, quanto a de valor. Na terceira forma, utilizamos o método `where()` e `orderBy()` antes de formar o vetor indexado resultante. Já na última forma, filtramos os resultados de uma maneira um pouco diferente. Assim, vimos como é possível, com poucas linhas de código, produzir uma lista de valores simples com base nos registros da base de dados.

app/control/dbsamples/CollectionArray.class.php

```

<?php
class CollectionArray extends TPage
{
    public function __construct()
    {
        parent::__construct();
        try {
            TTransaction::open('samples'); // abre uma transação

            // carrega um vetor indexado com todos os produtos
            $products = Product::getIndexedArray('id', 'description');
            var_dump($products);

            // carrega um vetor indexado com todos produtos, utilizando máscaras
            $products = Product::getIndexedArray('key:{id}',
                                                'description:{description}');
            var_dump($products);

            // carrega um vetor indexado com os produtos filtrados e ordenados
            $products = Product::where('unity', '=', 'PC')
                ->orderBy('id')
                ->getIndexedArray('id', 'description');
            var_dump($products);

            // carrega um vetor indexado com os produtos filtrados e ordenados
            $products = Product::where('id', 'in', [1,2,3,4])
                ->orderBy('id')
                ->getIndexedArray('id', 'description');
            var_dump($products);

            TTransaction::close(); // fecha a transação.
        }
        catch (Exception $e) {
            new TMessage('error', $e->getMessage());
        }
    }
}

```

3.4.7 Agregações

Quando desenvolvemos aplicações seguidamente precisamos agregar dados. Agregações permitem-nos realizar somas, médias, observando um agrupamento. Gráficos e relatórios frequentemente precisam apresentar dados agragados.

No exemplo a seguir produzimos diversos tipos diferentes de agregações. Os exemplos que seguem iniciam com uma soma simples da coluna total (**sumBy**), que retorna um número. Em seguida, realizamos uma contagem distinta da coluna total (**countDistinctBy**), que também retorna um número. Em seguida, realizamos a soma da coluna (**sumBy**), dentro de um agrupamento por data e cliente (**groupBy**), o que retorna um vetor com os resultados agrupados.

Ao realizarmos agregações, também podemos utilizar o método **where()** antes dos métodos de agrupamento (**groupBy**), para realizar filtragem dos dados antes do agrupamento. Quando utilizamos somente métodos de agregação (**sumBy**, **maxBy**) precedidos opcionalmente do método **where()**, teremos como retorno um número. Quando utilizamos o método de agrupamento **groupBy()**, o resultado será um vetor de objetos.

app/control/dbsamples/CollectionAggregation.class.php

```
<?php
class CollectionAggregation extends TPage
{
    public function __construct()
    {
        parent::__construct();
        try
        {
            TTransaction::open('samples');
            TTransaction::setLogger(new TLoggerSTD);

            echo '<pre>';

            var_dump(Sale::sumBy('total'));
            var_dump(Sale::countDistinctBy('total'));
            var_dump(Sale::groupBy('date', 'customer_id')->sumBy('total', 'total_alias'));
            var_dump(Sale::where('date', '>', '2015-03-12')
                ->sumBy('total'));
            var_dump(Sale::where('date', '>', '2015-04-12')
                ->groupBy('date')
                ->countDistinctBy('id', 'distinct_values'));
            var_dump(Sale::where('date', '>', '2015-03-12')
                ->groupBy('date')
                ->maxBy('total', 'max_value'));
            var_dump(Sale::where('date', '>', '2015-04-12')
                ->where('date', '<', '2019-04-12')
                ->sumBy('total'));
            var_dump(Sale::where('date', '>', '2015-04-12')
                ->where('date', '<', '2019-04-12')
                ->groupBy('customer_id')
                ->sumBy('total'));

            echo '</pre>';

            TTransaction::close();
        }
        catch (Exception $e)
        {
            new TMessage('error', $e->getMessage());
        }
    }
}
```

3.4.8 Alteração de coleções

A partir do momento em que temos uma coleção de objetos em memória, podemos manipulá-la. No exemplo anterior, carregamos uma coleção de objetos e, para cada objeto nesta coleção, exibimos algumas propriedades. Neste novo exemplo, no lugar de exibir alguma propriedade, alteraremos um atributo do objeto.

O exemplo inicia com a abertura da transação com a base de dados. Em seguida definimos um critério para obter todos os objetos em que o código da cidade (`city_id`) é igual a 4. Em seguida, é criado o repositório para manipular `Customer` e são carregados os objetos (`$customers`), conforme o critério definido. Após, percorremos cada um dos objetos, alterando seu atributo `phone`, definindo um novo prefixo e em seguida armazenando o Active Record por meio do método `store()`.

Obs: Uma vez que cada objeto da coleção é um Active Record, podemos usar dentro do laço de repetições métodos como: `store()` e `delete()` da classe `TRecord`.

app/control/dbsamples/CollectionUpdate.class.php

```
<?php
class CollectionUpdate extends TPage
{
    public function __construct()
    {
        parent::__construct();
        try
        {
            // abre uma transação
            TTransaction::open('samples');

            $criteria = new TCriteria;
            $criteria->add(new TFilter('city_id', '=', '4'));

            $repository = new TRepository('Customer');
            $customers = $repository->load($criteria);

            foreach ($customers as $customer)
            {
                $customer->phone = '84 '.substr($customer->phone, 3);
                $customer->store();
            }
            new TMessage('info', 'Registros atualizados');
            // fecha a transação.
            TTransaction::close();
        }
        catch (Exception $e)
        {
            new TMessage('error', $e->getMessage());
        }
    }
}
```

Você deve ter percebido por um lado que o programa funcionou adequadamente, mas, por outro lado, que os objetos foram atualizados um a um. Existem casos em que precisamos simplesmente substituir um atributo em uma grande quantidade de objetos de uma única vez. Neste caso, podemos realizar um UPDATE em lote.

No exemplo a seguir, criamos um critério para selecionar todos os clientes cujo id está entre `(1,2,3,4)`, e o status é `C` (Committed). A partir do critério criado, utilizamos o método `update()` da classe `TRepository`, para efetuar um UPDATE em lote. Este método recebe um vetor de valores a serem atualizados, e o próprio critério de filtro como segundo parâmetro. O resultado é um único comando `UPDATE`. O primeiro parâmetro gerará a cláusula `SET`, e o segundo, a cláusula `WHERE`.

app/control/dbsamples/CollectionBatchUpdate.class.php

```
<?php
class CollectionBatchUpdate extends TPage
{
    public function __construct()
    {
        parent::__construct();
        try {

```

```

TTransaction::open('samples');

$repos = new TRepository('Customer');
$criteria = new TCriteria;
$criteria->add(new TFilter('id', 'IN', [1,2,3,4]));
$criteria->add(new TFilter('status', '=', 'C'));

$values = array('gender' => 'F');
$repos->update($values, $criteria);

new TMessage('info', 'Records updated');
TTransaction::close();

}

catch (Exception $e)
{
    new TMessage('error', $e->getMessage());
}
}
}

```

O próximo exemplo é uma variação do anterior, para demonstrar de uma maneira ainda mais fácil como realizar a atualização em lote. Neste caso, estamos utilizando diretamente o método `where()` sobre a classe `Customer`, para aplicar um filtro de seleção. Cada chamada do `where()`, adiciona um filtro. Em seguida, executamos o método `set()`, para definir os novos valores. Cada chamada do `set()`, define o valor de um atributo. Por fim, o método `update()`, realiza a atualização em uma única vez.

`app/control/dbsamples/CollectionStaticBatchUpdate.class.php`

```

<?php
class CollectionStaticBatchUpdate extends TPage
{
    public function __construct()
    {
        parent::__construct();
        try
        {
            TTransaction::open('samples');

            Customer::where('id', 'IN', [1,2,3,4])
                ->where('status', '=', 'C')
                ->set('gender', 'M')
                ->update();

            new TMessage('info', 'Records updated');
            TTransaction::close();
        }
        catch (Exception $e)
        {
            new TMessage('error', $e->getMessage());
        }
    }
}

```

No exemplo anterior, os novos valores definidos pelo método `set()` eram fixos. Mas, em muitos alguns casos, é necessário realizar `UPDATE` alterando o valor de uma coluna em relação a ela mesma, ou em relação às outras. Neste caso, pode ser utilizado o prefixo `NOESC` no valor, para que não se adicione aspas ao redor. Neste exemplo, o preço de alguns produtos será reajustado em 20% em relação a ele mesmo.

```

Product::where('id', 'IN', [1,2,3,4])
    ->where('unity', '=', 'PC')
    ->set('sale_price', 'NOESC:sale_price * 1.2')
    ->update();

/*
UPDATE product
    SET sale_price = sale_price * 1.2
    WHERE (id IN (1,2,3,4) AND unity = 'PC')
*/

```

3.4.9 Exclusão de coleções

Das operações básicas, faltou vermos como apagar uma coleção de objetos. Neste próximo exemplo, apagaremos todos os clientes que residem na “Rua Porto” e que são do gênero masculino (`gender='M'`). Para tal, utilizamos a classe `TCriteria` para formar o filtro de seleção da mesma forma que nos exemplos anteriores. Para excluir os registros, utilizamos o método `delete()` da classe `TRepository`. Este método exclui os objetos com uma única instrução SQL sem carregá-los em memória.

app/control/dbsamples/CollectionDelete.class.php

```

<?php
class CollectionDelete extends TPage
{
    public function __construct()
    {
        parent::__construct();
        try
        {
            // abre uma transação
            TTransaction::open('samples');

            $criteria = new TCriteria;
            $criteria->add(new TFilter('address', 'like', 'Rua Porto%'));
            $criteria->add(new TFilter('gender', '=', 'M'));
            $repository = new TRepository('Customer');
            $repository->delete($criteria);

            new TMessage('info', 'Registros excluídos');

            // fecha a transação.
            TTransaction::close();
        }
        catch (Exception $e)
        {
            new TMessage('error', $e->getMessage());
        }
    }
}

```

O exemplo a seguir apresenta uma forma alternativa ainda mais simplificada para realizar a exclusão de objetos. A partir do método estático `where()` da classe `TRecord`, podemos estabelecer uma cadeia de instruções `where()`, onde cada cláusula é adicionada ao critério por um operador de `AND` (a não ser que o quarto parâmetro, que é opcional, seja informado). Ao fim da cadeia de `where()`, podemos executar o método `delete()`, excluindo os registros encontrados.

app/control/dbsamples/CollectionStaticSimpleDelete.class.php

```
<?php
class CollectionStaticSimpleDelete extends TPage
{
    public function __construct()
    {
        parent::__construct();
        try
        {
            TTransaction::open('samples');

            // exclui os objetos conforme o filtro
            Customer::where('address', 'like', 'Rua Porto%')
                ->where('gender', '=', 'M')
                ->delete();

            new TMessage('info', 'Records Deleted');
            TTransaction::close();
        }
        catch (Exception $e)
        {
            new TMessage('error', $e->getMessage());
        }
    }
}
```

3.4.10 Transformação de coleções

Aprendemos como carregar objetos da base de dados para a memória de diversas maneiras. Em muitas situações, teremos ainda que manipular estes objetos antes de passá-los adiante, seja para apresentação em tela, ou para outro método. Para estes casos, podemos percorrer os objetos com um simples FOREACH, ou também podemos utilizar o método **transform()**, encadeado após o **where()**. Este método recebe uma função anônima. Esta função anônima recebe o objeto (**\$object**) e tem o poder de modificá-lo. A lista final (**\$list**) já conterá estas modificações.

app/control/dbsamples/CollectionTransform.php

```
<?php
class CollectionTransform extends TPage
{
    public function __construct()
    {
        parent::__construct();
        try {
            TTransaction::open('samples');

            // carrega os clientes
            $list = Customer::where('id', '>', 1)->transform( function($object) {
                $object->name = 'Caro '. strtoupper($object->name);
                $object->address = strtoupper($object->address);
            });

            foreach ($list as $customer)
            {
                echo $customer->id . ' - ' .
                    $customer->name . ' - ' .
                    $customer->address . '<br>';
            }
            TTransaction::close(); // fecha a transação.
        }
    }
}
```

```

        catch (Exception $e) {
            new TMessage('error', $e->getMessage());
        }
    }
}

```

3.5 Relacionamentos entre objetos

3.5.1 Associação

Associação é um relacionamento entre dois objetos onde um objeto possui uma referência para outro. Assim, o objeto que possui a referência pode solicitar ao objeto associado a realização de ações por meio da invocação de métodos, ou mesmo obter atributos. Dentro do modelo utilizado neste capítulo, a relação de associação ocorre dentre outros, entre objetos da classe **Customer** e **Category** e também entre **Customer** e **City**, além de **City** e **State**. Como a relação é direcional, um objeto da classe **Customer** possui uma referência para um objeto **Category** e também **City**.

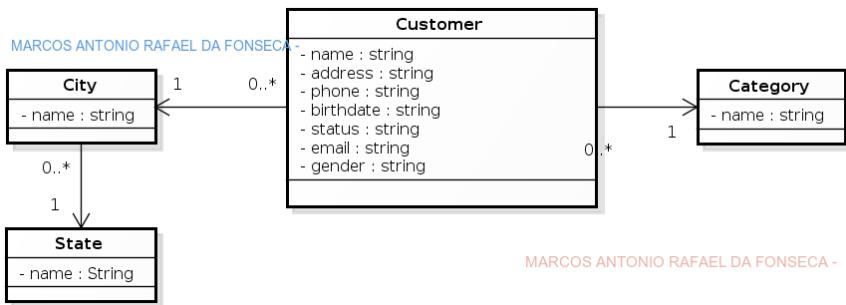


Figura 10 Exemplo de associação

Quando mapeado para um banco de dados relacional, o relacionamento de associação geralmente é transformado em pares de chaves primárias e estrangeiras. Em memória, temos um objeto **Customer** com uma referência para um objeto **Category**, então, em banco de dados teremos um registro na tabela **customer** com uma chave estrangeira (**category_id**) para a tabela **category**.

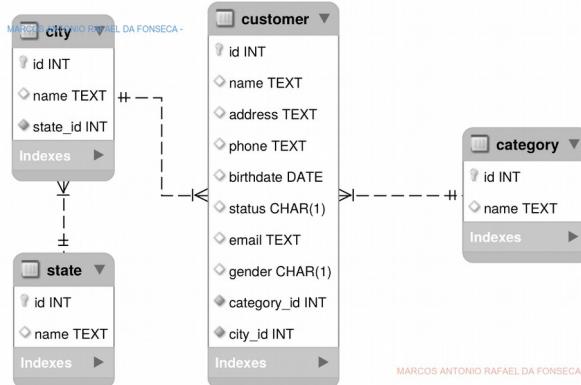


Figura 11 Uma associação mapeada para o modelo relacional

A implementação inicia com a classe Active Record para manipular **Category**.

app/model/Category.class.php

```

<?php
/**
 * Active Record para Category (categorias)
 */
class Category extends TRecord
{
    const TABLENAME = 'category';
    const PRIMARYKEY= 'id';
}
  
```

Normalmente em um relacionamento de associação, existe um método para atribuir o objeto relacionado (*setter*) e um para obter (*getter*). Após criarmos o Active Record para **Category**, alteramos a classe **Customer** para adicionar métodos *getters* e *setters* que permitirão atribuir e, também retornar um objeto **Category** (categoria) para determinado **Customer** (cliente).

O método **set_category()** será o método da classe **Customer** responsável por receber uma instância de **Category** e armazená-la internamente. Neste caso, é armazenado o próprio objeto **\$this->category** e também o seu ID **\$this->category_id**, que é armazenado no banco de dados quando o objeto é persistido. Já o método **get_category()** retorna a categoria atual do cliente. Se o objeto **\$this->category** ainda não existe, este método o cria, baseado no ID da categoria (**category_id**).

app/model/Customer.class.php

```
<?php
class Customer extends TRecord
{
    const ...

    private $category;

    // ... métodos já desenvolvidos ...
    /**
     * Atribui uma categoria
     * @param $category Objeto da classe Category
     */
    public function set_category(Category $category)
    {
        $this->category = $category; // armazena o objeto
        $this->category_id = $category->id; // armazena o ID do objeto
    }

    /**
     * Retorna a categoria associada
     * @return Objeto da classe Category
     */
    public function get_category()
    {
        if (empty($this->category))
        {
            $this->category = new Category($this->category_id);
        }

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

A partir dos métodos de associação, podemos acessar informações relacionadas à categoria pelo formato `$customer->category->name`. Sempre que acessarmos o atributo `category`, o método `get_category()` será executado.

```
<?php
class TestaAssociacao extends TPage
{
    public function __construct()
    {
        parent::__construct();
        try
        {
            TTransaction::open('samples'); // abre uma transação
            $customer= new Customer(4); // carrega o cliente 4
            echo $customer->category->name; // chama indiretamente get_category()
            TTransaction::close(); // fecha a transação.
        }
        catch (Exception $e)
        {
            new TMessage('error', $e->getMessage());
        }
    }
}
```

3.5.2 Composição

A composição é um relacionamento conhecido como todo/parte. É um relacionamento entre dois objetos onde um objeto (todo) é formado por uma ou mais partes. Neste relacionamento cada uma das partes é exclusiva do objeto todo, ou seja, uma parte não pode pertencer a (ser compartilhada com) outros objetos. No modelo de classes, a composição aparece no relacionamento entre **Customer** e **Contact**, sendo que um cliente pode ser composto por vários objetos do tipo **Contact**, e cada objeto **Contact** pertence apenas a um **Customer**.

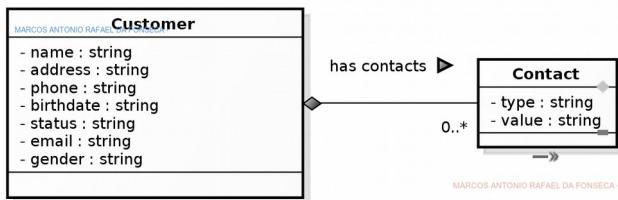


Figura 12 Relacionamento de composição

A composição pode ser mapeada para o banco de dados na forma de uma chave estrangeira na tabela que representa a parte (**contact**), apontando para a tabela que representa o todo (**customer**), uma vez que a parte é exclusiva do todo. Na figura seguinte, podemos ver o modelo relacional resultante da composição. A tabela **contact**, que representa a classe **Contact**, possui uma chave estrangeira apontando para a tabela **customer**, que representa **Customer**.

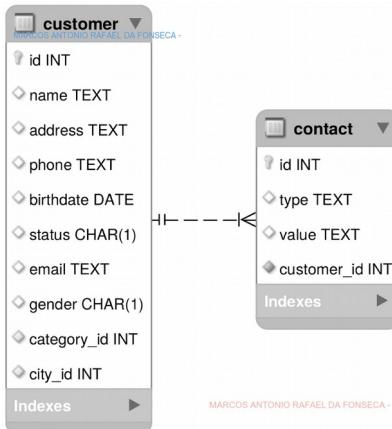


Figura 13 Uma composição mapeada para o modelo relacional

A implementação inicia com uma classe Active Record para manipular **Contact**.

app/model/Contact.class.php

```
<?php
class Contact extends TRecord
{
    const TABLENAME = 'contact';
    const PRIMARYKEY= 'id';
}
```

O próximo passo é alterar a classe `Customer`, que representa o “todo” na relação. Esta classe deverá prover métodos para manipular a sua relação com `Contact`. Para tal, serão criados os métodos: `clearParts()`, responsável por eliminar os objetos que representam as partes da relação (instâncias de `Contact`); `addContact()`, que receberá um objeto `Contact` e integrará este ao objeto `Customer` na forma de um vetor (`$this->contacts`); e `getContacts()`, que retornará as partes (objetos `Contact`).

app/model/Customer.class.php

```
<?php
class Customer extends TRecord
{
    const ...
    private $contacts;
    // ... métodos já desenvolvidos ...

    public function clearParts()
    {
        $this->contacts = array();
    }

    /**
     * Adiciona um contato
     * @param $contact Objeto da classe Contact
     */
    public function addContact(Contact $contact)
    {
        $this->contacts[] = $contact;
    }

    /**
     * Retorna um vetor de contatos
     * @return Vetor de objetos da classe Contact
     */
    public function getContacts()
    {
        return $this->contacts;
    }
}
```

Agora que já escrevemos os métodos básicos para relacionar o objeto `Customer` com o objeto `Contact` por meio de composição, também devemos pensar na persistência. E o que muda para um objeto `Customer` em relação à persistência? Normalmente a persistência de um objeto Active Record é individualizada, ou seja, não influencia outros objetos. Mas em relações todo/parte, é interessante que no momento em que salvamos, carregamos ou excluímos um objeto Active Record que represente o “todo” de uma relação, também suas partes sejam salvas, carregadas ou excluídas.

Para que as partes sejam persistidas junto com o objeto “todo”, podemos sobrecarregar métodos como `load()`, `store()` e `delete()`, que são os três métodos básicos para persistência de um Active Record. Assim, os métodos se comportarão da seguinte maneira:

- `load()`: Carregará um Active Record e também os seus objetos compostos;
- `store()`: Armazenará um Active Record e os seus objetos compostos;
- `delete()`: Excluirá um Active Record e também os seus objetos compostos.

Começaremos a alteração pelo método `load()`. Neste método, responsável pela carga de objetos, antes de realizarmos a carga do objeto em si (`parent::load($id)`), carregaremos as suas partes. Para tal, utilizaremos o método `load()` para carregar os contatos, que são armazenados na propriedade `contacts`. Os objetos `Contact` são carregados por meio da identificação de sua chave estrangeira (`customer_id`).

app/model/Customer.class.php

```
<?php
class Customer extends TRecord
{
    // ... conteúdo já desenvolvido ...

    public function load($id)
    {
        $this->contacts = Contact::where('customer_id', '=', $id)->load();
        return parent::load($id); // carrega o próprio objeto
    }
}
```

Outra maneira de alcançar o mesmo objetivo, que é de carregar os objetos da composição, é por meio do método `loadComposite()`. Este método recebe como parâmetros a classe composta (`Contact`), a chave estrangeira de ligação, e o ID da classe principal, e retorna todos os objetos vinculados ao registro.

```
public function load($id)
{
    $this->contacts = parent::loadComposite('Contact', 'customer_id', $id);
    return parent::load($id); // carrega o próprio objeto
}
```

O método `store()`, responsável pela persistência de um objeto também será alterado. Neste caso, após a persistência do objeto em si (`parent::store()`), cada uma de suas partes são persistidas. Para armazenar as partes (`Contact`), primeiramente procedemos com a exclusão dos contatos já existentes relativos àquele cliente. Para excluir as partes, utilizamos o método `delete()`, identificando a chave estrangeira (`customer_id`). Após excluir, percorremos todas as partes (objetos `Contact`) em memória, atribuímos a eles o ID do cliente em sua chave estrangeira (`customer_id`) e armazenamos estes objetos por meio do método `store()`.

```
public function store()
{
    parent::store();
    Contact::where('customer_id', '=', $this->id)->delete();
```

```

if ($this->contacts) {
    foreach ($this->contacts as $contact)
    {
        $contact->customer_id = $this->id;
        $contact->store();
    }
}
}

```

Outra forma de alcançar o mesmo objetivo é por meio do método `saveComposite()`, que armazena todos os objetos vinculados. Este método recebe como parâmetros: a classe vinculada (`Contact`); a chave estrangeira de ligação; o ID do registro principal; e, o vetor contendo as partes (`$this->contacts`).

```

public function store()
{
    parent::store(); // armazena o próprio objeto
    parent::saveComposite('Contact', 'customer_id', $this->id, $this->contacts);
}

```

Por fim, o método `delete()` também é sobrecarregado. Dessa forma, sempre que excluirmos o objeto “todo”, também as suas partes serão excluídas. O método `delete` recebe opcionalmente o código do objeto a ser excluído (se não receber, usa `$this->id`). Assim, o método realiza primeiro a exclusão das partes (objetos `Contact`), por meio do método `delete()`, identificando como filtro do critério o código do cliente (`customer_id`). Após excluir as partes, procedemos com a exclusão do objeto cliente em si por meio do método `delete()` da classe `TRecord` (`parent::delete($id)`).

```

public function delete($id = NULL)
{
    $id = isset($id) ? $id : $this->id;

    Contact::where('customer_id', '=', $id)->delete();
    parent::delete($id); // apaga o próprio objeto
}

```

Outra forma de alcançar o mesmo objetivo é por meio do método `deleteComposite()`, que recebe como parâmetros: a classe vinculada (`Contact`); a chave de ligação (`customer_id`); e o registro principal; e, exclui os objetos vinculados.

```

public function delete($id = NULL)
{
    parent::deleteComposite('Contact', 'customer_id', $id);
    parent::delete($id); // apaga o próprio objeto
}

```

A decisão de carregar ou persistir as partes de um objeto de maneira automática sempre que o objeto “todo” é carregado ou persistido é chamada de *eager loading* e deve ser tomada caso a caso. Esta nem sempre é a melhor solução, pois em muitos casos carregaremos uma série de objetos em memória desnecessariamente, impactando a performance. Outra possibilidade de implementação é deixarmos os métodos `load()` e `store()` intactos, e criarmos um método para persistir, e outro para retornar as partes do objeto separadamente. Assim, ao salvarmos ou carregarmos o objeto “todo”, não estariam realizando estas operações também com suas “partes”.

```

public function addContact(Contact $contact)
{
    $contact->customer_id = $this->id;
    $contact->store();
}

public function getContacts()
{
    return Contact::where('customer_id', '=', $this->id)->load();
}

```

O próximo exemplo demonstra o funcionamento dos métodos criados anteriormente. Para tal, abrimos uma transação com a base de dados, carregamos o cliente de ID 4 para a memória e em seguida adicionamos a este objeto dois contatos (objetos **Contact**). No fim, salvamos o objeto **Customer** (cliente) na base de dados (**store**). Neste momento, os dois objetos são inseridos na tabela **contact**, com a chave **customer_id** apontando para o cliente 4 (objeto **Customer**).

app/control/dbsamples/ObjectComposition.class.php

```

<?php
class ObjectComposition extends TPage
{
    public function __construct()
    {
        parent::__construct();
        try
        {
            TTransaction::open('samples'); // abre uma transação
            $customer= new Customer(4); // carrega o cliente 4

            // cria dois objetos Contact
            $contact1 = new Contact;
            $contact2 = new Contact;

            // define o valor de alguns atributos
            $contact1->type  = 'fone';
            $contact1->value = '78 2343-4545';

            $contact2->type  = 'fone';
            $contact2->value = '78 9494-0404';
            // adiciona os contatos ao cliente
            $customer->addContact($contact1);
            $customer->addContact($contact2);

            $customer->store(); // armazena o cliente (e seus contatos)

            new TMessage('info', 'Contatos adicionados');
            TTransaction::close(); // fecha a transação
        }
        catch (Exception $e)
        {
            new TMessage('error', $e->getMessage());
        }
    }
}

```

3.5.3 Agregação

A agregação também é um relacionamento todo/parte, tal qual a composição. Assim, também temos um objeto “todo” composto de uma ou mais “partes”. A diferença primordial entre os dois tipos de relacionamento é que na agregação, um objeto “parte” não é exclusivo do “todo”, ou seja, pode pertencer à diferentes objetos “todo”. A figura a seguir mostra o exemplo de agregação utilizado neste livro. Neste exemplo, temos objeto **Customer** (cliente) com uma agregação com o objeto **Skill** (habilidades).

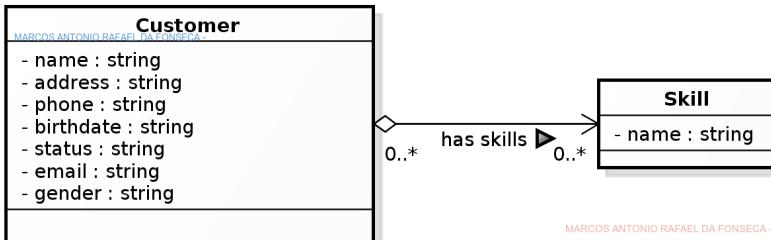


Figura 14 Relacionamento de agregação

Como vimos, na agregação um objeto “parte” pode estar vinculado à diferentes objetos “todo”. Neste caso em específico, um cliente pode ter diferentes habilidades (esportes, literatura, música) e uma habilidade pode pertencer à diferentes clientes. Dessa forma, não é possível representar essa estrutura em banco de dados apenas com duas tabelas. Assim, surge a necessidade de uma terceira tabela, chamada de tabela associativa (*Association Table Mapping*). O objetivo desta tabela é armazenar pares de chaves estrangeiras. Nesta tabela, cada linha terá uma chave estrangeira para a tabela **customer** e outra para a tabela **skill**. Assim, poderemos ter vários registros dos relacionamentos entre objetos **Customer** e objetos **Skill**.

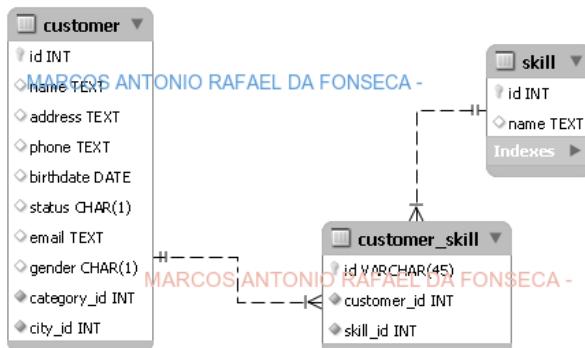


Figura 15 Uma agregação mapeada para o modelo relacional

Um cliente (`Customer`) pode ter uma ou mais habilidades (`Skill`). É necessário criar um Active Record para manipular objetos de `Skill`. Além disso, precisaremos de um Active Record para manipular os registros da tabela associativa `customer_skill`. Para tal, vamos criar o Active Record `CustomerSkill`.

app/model/Skill.class.php

```
<?php
/**
 * Active Record para Skill (habilidades)
 */
class Skill extends TRecord
{
    const TABLENAME = 'skill';
    const PRIMARYKEY= 'id';
}
```

app/model/CustomerSkill.class.php

```
<?php
/**
 * Active Record para agregação entre Customer e Skill
 */
class CustomerSkill extends TRecord
{
    const TABLENAME = 'customer_skill';
    const PRIMARYKEY= 'id';
}
```

Agora que já temos as classes criadas, é necessário alterar a classe `Customer`, que representa o “todo” na relação, para que possamos acrescentar os objetos `Skill` (partes). Para tal, vamos modificar o método `clearParts()`, que limpa todas as partes do objeto; criar o método `addSkill()`, que acrescenta uma habilidade; e criar o método `getSkills()`, que retorna as habilidades (`Skill`).

app/model/Customer.class.php

```
<?php
class Customer extends TRecord
{
    const ...
    private $skills;
    // ... métodos já desenvolvidos ...

    public function clearParts()
    {
        $this->skills = array();
    }

    public function addSkill(Skill $skill)
    {
        $this->skills[] = $skill;
    }

    public function getSkills()
    {
        return $this->skills;
    }
}
```

Com a última alteração, podemos adicionar objetos `Skill` a um objeto `Customer`. Entretanto, é necessário alterar os métodos persistência: `load()`, `store()` e `delete()` para que no momento de persistir um cliente (`Customer`), também sejam persistidas as suas partes (`Skill`).

Inicialmente vamos alterar o método `load()`, para que as habilidades (`Skill`) vinculadas a um cliente (`Customer`) sejam carregadas toda vez que um objeto `Customer` for carregado. O método `load()` já foi alterado uma vez para que os contatos fossem carregados com o objeto cliente. Agora vamos acrescentar algumas linhas para carregar as habilidades.

As habilidades são carregadas a partir da classe que representa a tabela associativa (`CustomerSkill`). Assim, vamos carregar todos os registros dessa tabela associativa, filtrados pelo id do cliente. Para cada registro que retornar, vamos instanciar o objeto `Skill` correspondente (`new Skill`) e acrescentar este objeto ao vetor de habilidades, por meio do método `addSkill()`. Assim, quando um cliente (`Customer`) for carregado em memória, também os objetos `Skill` agregados serão.

Obs: As partes dos métodos `load()`, `store()` e `delete()` correspondentes à persistência dos contatos foi cortada, mas encontram-se completas no Tutor.

app/model/Customer.class.php

```
<?php
class Customer extends TRecord
{
    const ...
    private $skills;

    // métodos já desenvolvidos...

    public function load($id)
    {
        // carrega as habilidades do cliente a partir de CustomerSkill
        $customer_skills = CustomerSkill::where('customer_id', '=', $id)->load();

        if ($customer_skills)
        {
            // percorre os objetos CustomerSkill
            foreach ($customer_skills as $customer_skill)
            {
                // adiciona a habilidade ao cliente
                $this->addSkill( new Skill($customer_skill->skill_id) );
            }
        }

        return parent::load($id); // carrega o próprio objeto Customer
    }
}
```

Outra forma de atingir o mesmo objetivo, é por meio do método `loadAggregate()`, que recebe parâmetros como: a classe agregada; a classe associativa; o campo de ligação para o registro principal; o campo de ligação para o registro agregado; e o ID do registro principal, e carrega os objetos agregados.

```
public function load($id)
{
    $this->skills = parent::loadAggregate('Skill', 'CustomerSkill', 'customer_id',
                                         'skill_id', $id);
    return parent::load($id); // carrega o próprio objeto
}
```

O próximo passo é alterar o método `store()`, para que no momento de persistir um cliente (`Customer`), também as suas habilidades (`Skill`) relacionadas sejam persistidas. Para isso, inicialmente vamos armazenar o próprio cliente (`parent::store()`), e em seguida excluir as suas habilidades já registradas (`CustomerSkill`), para depois armazenar as habilidades que estão em memória (`$this->skills`).

```
public function store()
{
    parent::store(); // armazena o próprio cliente
    // exclui as habilidades existentes na base de dados
    CustomerSkill::where('customer_id', '=', $this->id)->delete();
    if ($this->skills) {
        // percorre as habilidades em memória
        foreach ($this->skills as $skill) {
            // cria um objeto CustomerSkill e persiste-o
            $customer_skill = new CustomerSkill;
            $customer_skill->customer_id = $this->id;
            $customer_skill->skill_id = $skill->id;
            $customer_skill->store();
        }
    }
}
```

Veja que não apagamos as habilidades em si (objeto `Skill`), mas somente os registros que ligam o cliente às suas habilidades (`CustomerSkill`), que estão armazenados na tabela `customer_skill`. Após, percorremos as habilidades (`Skill`) do cliente (`$this->skills`) e persistimos uma a uma na forma de um objeto `CustomerSkill`, sendo que o mesmo possui uma referência para o cliente (`$this->id`) e outra para a habilidade (`$skill->id`).

Outra forma de atingir o mesmo objetivo é por meio do método `saveAggregate()`, que recebe parâmetros como: a classe associativa, a chave de ligação para o registro principal, a chave para o registro agregado, o ID do registro principal, e os objetos em si (`skills`), e armazena estes em `CustomerSkill`.

```
public function store()
{
    parent::store(); // armazena o próprio objeto
    parent::saveAggregate('CustomerSkill', 'customer_id', 'skill_id',
                          $this->id, $this->skills);
}
```

Por fim, lembramos que sempre que um cliente (`Customer`) é excluído, também devem ser excluídas as habilidades relacionadas a ele. Perceba que o objetivo aqui não é excluir da base de dados o registro de uma habilidade (`Skill`), mas somente o vínculo que liga um cliente (`Customer`) a uma habilidade (`Skill`), representado pelo objeto `CustomerSkill`.

O método `delete()` será sobrecarregado para apagar, além do próprio cliente (`parent::delete()`), os objetos relacionados em `CustomerSkill`. Desta forma, excluiremos todos os vínculos do cliente a objetos `Skill`, antes de excluir o próprio cliente.

```
public function delete($id = NULL)
{
    $id = isset($id) ? $id : $this->id;

    CustomerSkill::where('customer_id', '=', $id)->delete();
    parent::delete($id); // apaga o próprio objeto
}
```

Outra forma de atingir o mesmo objetivo, é por meio do método `deleteComposite()`, que exclui um conjunto de objetos relacionados. Neste caso, estamos excluindo todos os objetos da classe `CustomerSkill`, por meio do campo de ligação `customer_id`, identificando também o ID do registro principal.

```
public function delete($id = NULL)
{
    $id = isset($id) ? $id : $this->id;

    parent::deleteComposite('CustomerSkill', 'customer_id', $id);
    parent::delete($id); // apaga o próprio objeto
}
```

Para demonstrar a agregação em funcionamento, foi criado o exemplo a seguir. Neste exemplo, carregamos em memória o cliente 4 e em seguida adicionamos a este cliente duas habilidades: 1 e 2. Na agregação, adicionamos em um objeto “todo” (neste caso `Customer`) objetos já existentes. Ao final o cliente é persistido e neste passo também são persistidas as suas ligações com essas duas habilidades.

`app/control/dbsamples/ObjectAggregation.class.php`

```
<?php
class ObjectAggregation extends TPage
{
    public function __construct()
    {
        parent::__construct();
        try {
            TTransaction::open('samples'); // abre uma transação
            $customer= new Customer(4); // carrega o cliente 4

            // adiciona habilidades
            $customer->addSkill(new Skill(1));
            $customer->addSkill(new Skill(2));

            // persiste o cliente
            $customer->store();

            new TMessage('info', 'Habilidades adicionadas');
            TTransaction::close(); // fecha a transação.
        }
        catch (Exception $e) {
            new TMessage('error', $e->getMessage());
        }
    }
}
```

A decisão de carregar ou persistir as partes de um objeto de maneira automática sempre que o objeto “todo” é carregado ou persistido é chamada de *eager loading* e deve ser tomada caso a caso. Esta nem sempre é a melhor solução, pois em muitos casos carregaremos uma série de objetos em memória desnecessariamente, impactando a performance. Outra possibilidade de implementação é deixarmos os métodos `load()` e `store()` intactos, e criarmos um método para persistir, e outro para retornar as partes do objeto separadamente. Assim, ao salvarmos ou carregarmos o objeto “todo”, não estaríamos realizando estas operações também com suas “partes”.

```
public function addSkill(Skill $skill)
{
    $customer_skill = new CustomerSkill();
    $customer_skill->customer_id = $this->id;
    $customer_skill->skill_id     = $skill->id;
    $customer_skill->store();
}

public function getSkills()
{
    return parent::loadAggregate('Skill', 'CustomerSkill', 'customer_id',
                                  'skill_id', $this->id);
}
```

3.5.4 Atalhos em relacionamentos

Neste exemplo, vamos demonstrar formas variadas de obter dados relacionados sem a necessidade de implementar métodos de relacionamentos nas classes. Os métodos a seguir são nativos do próprio Framework e permitem acessar objetos relacionados.

Os primeiros quatro exemplos tratam de uma relação 1-N. Assim, eles buscam a partir de um cliente (`Customer`), vários registros de contatos (`Contact`). É importante lembrar que a tabela de contatos possui uma chave estrangeira para clientes.

O primeiro exemplo, carrega o cliente com um `find(1)`, e usa o método `hasMany()` para retornar os objetos da classe `Contact` associados. Neste formato, o Framework tenta utilizar um campo de chave estrangeira em `Contact` chamado `customer_id` (nome da classe em minúsculo, com sufixo `_id`). O segundo exemplo também carrega o cliente com um `find(1)`, mas usa uma versão mais completa do `hasMany()`, informando além da classe relacionada, o nome da chave estrangeira que relaciona as duas, o campo de chave primária, e o campo de ordenação. O terceiro exemplo, após carregar o cliente com um `find(1)`, filtra os contatos relacionados com o método `filterMany()`, seguido de um `where()`, assim, os contatos já vêm filtrados. O quarto exemplo, usa uma versão mais sofisticada do `filterMany()`, especificando a chave estrangeira da tabela de ligação, e em seguida filtrando os dados antes de retorná-los.

Os próximos exemplos tratam de uma relação N-N, visto que existe entre `Customer` e `Skill` uma relação `CustomerSkill`, representada também por uma tabela chamada `customer_skill` na base de dados.

O quinto exemplo, carrega o cliente com um `find(1)`, e os objetos relacionados da classe `Skill` (que é uma relação N-N). Neste caso, o Framework tentará buscar em uma

tabela chamada `customer_skill` (formada pela junção das classes) automaticamente todas as referências para `Skill`. O sexto exemplo é uma variação do quinto, em que é especificada não somente a classe relacionada (`Skill`), mas também a classe de ligação (`CustomerSkill`), e as chaves para ambas, uma vez que se trata de uma relação N-N, a tabela intermediária possui chaves para `Customer` e `Skill`.

[app/control/dbsamples/RelationShortcuts.class.php](#)

```
<?php

class RelationShortcuts extends TPage
{
    public function __construct()
    {
        parent::__construct();
        try {
            TTransaction::open('samples'); // abre uma transação

            // 1. Versão simplificada carrega os objetos Contato relacionados (1-N)
            $contacts = Customer::find(1)
                ->hasMany('Contact');
            var_dump($contacts);

            // 2. Versão completa, carrega os objetos Contato relacionados (1-N)
            $contacts = Customer::find(1)
                ->hasMany('Contact', 'customer_id', 'id', 'type');
            var_dump($contacts);

            // 3. Filtra os objetos Contato relacionados antes de retornar (1-N)
            $contacts = Customer::find(1)
                ->filterMany('Contact')
                ->where('type', '=', 'MSN')
                ->load();
            var_dump($contacts);

            // 4. Versão simplificada. Carrega objetos por tabela associativa (N-N)
            $skills = Customer::find(1)
                ->belongsToMany('Skill');
            var_dump($skills);

            // 5. Versão completa. Carrega objetos por tabela associativa (N-N)
            $skills = Customer::find(1)
                ->belongsToMany('Skill', 'CustomerSkill',
                    'customer_id', 'skill_id');
            var_dump($skills);

            TTransaction::close(); // fecha a transação.
        }
        catch (Exception $e) {
            new TMessage('error', $e->getMessage());
        }
    }
}
```

CAPÍTULO 4

Componentes de apresentação

O foco deste capítulo é explorar os componentes de apresentação do Framework. Portanto, vamos explorar tabelas, painéis e notebooks; componentes para criação de formulários e datagrids; diálogo de mensagens, dentre outros.

4.1 Conceitos básicos

4.1.1 Controlador de páginas

Qualquer elemento a ser apresentado pelo Framework deve estar contido em uma página. Para criarmos uma nova página, devemos criar uma classe. Assim, cada página a ser exibida no framework será uma classe. Existem basicamente dois controladores de páginas: **TPage** e **TWindow**. Qualquer página deve ser subclasse de um desses dois controladores padrão. Enquanto subclasses de **TPage** são exibidas dentro do layout do sistema, subclasses de **TWindow** são exibidas em uma janela (camada).

A seguir temos o exemplo de uma página simples contendo apenas uma mensagem de Hello World. No método construtor da página acrescentamos seu conteúdo, por meio do método **add()**. Aqui estamos utilizando o componente **TLabel**, que representa um rótulo de texto. Neste capítulo, abordaremos outros componentes visuais que podem ser utilizados.

[app/control/Presentation/HelloWorld.class.php](#)

```
<?php
class HelloWorld extends TPage
{
    public function __construct()
    {
        parent::__construct(); // é fundamental executar o construtor da classe pai
        parent::add(new TLabel('Hello World'));
    }
}
```

Para exibir um controlador, podemos editar o layout da aplicação (HTML) e acrescentar um link para `index.php`, identificando a classe que gostaríamos que o Framework exibisse por meio do parâmetro `class`:

```
<a href="index.php?class=HelloWorld" generator="adianti"> Hello World</a>
```

Obs: A presença do atributo `generator=adianti` é importante pois é ele que determina que o conteúdo do link seja carregado dinamicamente, por meio de uma requisição para o `engine.php`, carregando somente o núcleo da página, não causando uma recarga completa.

Ou também podemos simplesmente digitar no navegador:

```
http://localhost/tutor/index.php?class=HelloWorld
```

O resultado da execução pode ser visto na figura a seguir.



Figura 16 Página com Hello World

Também podemos utilizar o controlador de páginas `TWindow` para criar páginas. O conteúdo das subclasses de `TWindow` é exibido em janelas sobre o sistema. A seguir temos uma página subclasse de `TWindow` e o resultado de sua execução.

app/control/Presentation/HelloWorldWindow.class.php

```
<?php
class HelloWorldWindow extends TWindow
{
    public function __construct()
    {
        parent::__construct();
        parent::setTitle('Window title');
        parent::setSize(400, 200);
        parent::add(new TLabel('Hello World'));
    }
}
```



Figura 17 Janela com Hello World

Qualquer classe filha de **TPage** automaticamente terá seu conteúdo renderizado dentro do layout do sistema (**Layout.html**) na **<div>** com o ID **adianti_div_content**. Já qualquer classe filha de **TWindow** terá seu conteúdo renderizado dentro do layout do sistema (**layout.html**) dentro da **<div>** com o ID **adianti_online_content**.

Além de deixar que o Framework decida automaticamente o local em que o conteúdo de cada classe será carregado (conforme **TPage** ou **TWindow**), como visto anteriormente, é possível indicar exatamente em qual elemento do DOM será carregado o conteúdo de uma página. No exemplo a seguir, supomos que desejamos exibir o conteúdo da classe **MessageList** no elemento do DOM com id **message_area**. Neste caso, basta definir um atributo chamado **adianti_target_container**.

app/control/Presentation/Extras/MessageList.class.php

```
<?php
class MessageList extends TPage
{
    public function __construct()
    {
        parent::__construct();

        // O #id do container da página
        $this->adianti_target_container = 'message_area';
```

Obs: Como cada página é carregada por meio de requisições JavaScript assíncronas, poderemos ter diferentes páginas carregadas ao mesmo tempo, cada uma em seu container. No Tutor existe um exemplo chamado **InboxView**, que demonstra uma pequena caixa de correio eletrônico utilizando este conceito. Assim, quando o usuário clicar em Inbox (Entrada), ou Sent (enviados), apenas as mensagens são carregadas em um espaço no núcleo da tela.

4.1.2 Ações

Não desejamos construir toda a aplicação no seu método construtor, não é mesmo? O método construtor das páginas geralmente é utilizado para a construção do visual inicial da página e a ação desempenhada em uma página é mapeada como um método. No exemplo a seguir temos uma página com um método **onHelloWorld()**.

app/control/Presentation/HelloWorldAction.class.php

```
<?php
class HelloWorldAction extends TPage
{
    public function __construct()
    {
        parent::__construct();
    }

    public function onHelloWorld()
    {
        parent::add(new TLabel('Hello World'));
    }
}
```

Mas como solicitamos ao Framework para que ele execute um método de uma página? Na Web podemos editar o layout da aplicação e acrescentar um link para `index.php`, da seguinte forma:

```
<a href="index.php?class=HelloWorldAction&method=onHelloWorld"
    generator="adianti">Hello</a>
```

Ou também podemos simplesmente digitar no navegador:

```
http://localhost/tutor/index.php?class=HelloWorldAction&method=onHelloWorld
```

No exemplo anterior, deixamos a responsabilidade de adicionar o conteúdo à página dentro do método `onHelloWorld()`. Mas na maioria das vezes em que formos construir páginas mais complexas (Ex: formulários) nos próximos capítulos do livro, utilizaremos o método construtor para definir (construir) a interface. Mas isso não nos impede de modificar o conteúdo que já foi adicionado pelo método construtor posteriormente.

No exemplo a seguir, criamos um label (`TLabel`) com um conteúdo vazio já no método construtor, e adicionamos este à página. Quando acionarmos o método `onHelloWorld()`, o conteúdo do label será modificado pelo método `setValue()`, e isto acontecerá antes da página ser exibida. Para acionarmos esse método, basta digitarmos na URL o seguinte:

```
http://localhost/tutor/index.php?class=HelloWorldAction2&method=onHelloWorld
```

Para descobrirmos quais métodos podemos utilizar, basta consultar a documentação da API das classes do Framework, ou usar o recurso de autocomplete.

app/control/Presentation/HelloWorldAction2.class.php

```
<?php
class HelloWorldAction2 extends TPage
{
    private $label;

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

        // cria o label
        $this->label = new TLabel('');

        // adiciona o label à página
        parent::__add($this->label);
    }

    public function onHelloWorld()
    {
        // modifica o conteúdo do label
        $this->label->setValue('Hello World');
    }
}
```

4.2 Páginas

Nesta seção, será demonstrado como criar alguns tipos comuns de páginas, para exibição de HTML, PDF, sistemas externos e outros. Nas próximas seções, exploraremos os componentes do Framework para diálogos, formulários, datagrids, e outros.

4.2.1 Página com HTML

Neste primeiro exemplo, vamos demonstrar como criar uma classe que exibe uma simples página em HTML, com algumas substituições de variáveis. Para tal, vamos criar uma classe `SinglePageView` e utilizar o componente `THtmlRenderer` para realizar a interpretação de um template (`app/resources/page.html`). A classe `THtmlRenderer` permite ainda, passar conteúdos para o arquivo, por meio do vetor `$replaces`.

`app/control/Presentation/Pages/SinglePageView.class.php`

```
<?php
class SinglePageView extends TPage
{
    public function __construct()
    {
        parent::__construct();

        // processa um template de página em HTML
        $this->html = new THtmlRenderer('app/resources/page.html');

        $replaces = [];
        $replaces['title'] = 'Panel title';
        $replaces['footer'] = 'Panel footer';
        $replaces['name'] = 'Someone famous';

        // habilita a seção main, injetando o vetor de substituições
        $this->html->enableSection('main', $replaces);

        parent::add($this->html);
    }
}
```

Todo o template da página HTML deve estar contido entre as tags `<!--[main]-->` e `<!--[/main]-->`. As variáveis a serem substituídas pelo método `enableSection()` e pelo vetor `$replaces` devem estar informadas no formato `{$variável}`.

`app/resources/page.html`

```
<!--[main]-->
<div class="card">
    <div class="card-header">
        <div class="card-title"> {$title} </div>
    </div>
    <div class="card-body">
        <blockquote class="blockquote">
            <p>Lorem ipsum dolor sit amet...</p>
            <footer> <b>{$name}</b> in ...</footer>
        </blockquote>
    </div>
    <div class="card-footer"> {$footer} </div>
<!--[/main]-->
```

A figura a seguir demonstra a página criada com o conteúdo HTML.



Figura 18 Página simples com HTML

Obs: Procure sempre utilizar o recurso de templates para isolar o PHP do HTML. Códigos HTML misturados com códigos PHP prejudicam a legibilidade do sistema e não são considerados uma boa prática. Recursos mais elaborados sobre apresentação e processamento de HTML serão demonstrados no tópico “Templates”.

4.2.2 Página com PDF embutido

Este exemplo demonstra como exibir um arquivo PDF dentro de uma página. Para tal, criaremos manualmente um elemento (tag) do tipo `object`. Esta tag, a ser apresentada pelo HTML possui alguns atributos como: `data` (caminho do PDF), `type` (tipo do conteúdo exibido), e `style` (atributos altura e largura da apresentação CSS).

app/control/Presentation/Pages/EmbeddedPDFView.class.php

```
<?php
class EmbeddedPDFView extends TPage
{
    public function __construct()
    {
        parent::__construct();

        $object = new TElement('object');
        $object->data  = 'http://www.adianti.com.br/framework_files/adianti_framework.pdf';
        $object->type  = 'application/pdf';
        $object->style = "width: 100%; height:600px";
        parent::add($object);
    }
}
```

A figura a seguir demonstra o arquivo PDF sendo exibido.



Figura 19 Página simples com PDF

4.2.3 Página com conteúdo externo

Este exemplo demonstra como embarcar um site ou sistema externo dentro de uma página do Framework. Para tal, é necessário utilizar um elemento HTML `iframe` para isolar o conteúdo do convidado (site ou sistema externo) do hospedeiro (sistema feito com o Framework). Este isolamento não permite que bibliotecas CSS, Javascript, dentre outros entrem em conflito pois ficam em contextos diferentes. Apesar de visualmente parecer integrado ao sistema, é como se estivessem em janelas separadas.

Nem todo site ou sistema permite ser embarcado por um domínio externo. Se você tentar embarcar o Google, não funcionará e você verá uma mensagem do tipo “Refused to display 'https://www.google.com/' in a frame because it set 'X-Frame-Options' to 'sameorigin!'” no console Javascript. Já o site Wikipedia permite ser embarcado (no momento da escrita deste exemplo). Caso você deseja embarcar um sistema próprio e ele estiver no mesmo domínio, não terá problemas, mas se estiverem em domínios diferentes, pesquise sobre “CORS request (Cross-Origin Resource Sharing)”.

app/control/Presentation/Pages/ExternalPageView.class.php

```
<?php
class ExternalPageView extends TPage
{
    public function __construct()
    {
        parent::__construct();

        $iframe = new TElement('iframe');
        $iframe->id      = "iframe_external";
        $iframe->src     = "https://pt.wikipedia.org/wiki/PHP";
        $iframe->frameborder = "0";
        $iframe->scrolling = "yes";
        $iframe->width   = "100%";
        $iframe->height  = "700px";
        parent::add($iframe);
    }
}
```

A figura a seguir demonstra a página com o site Wikipedia embarcado.



Figura 20 Página exibindo site Wikipedia

4.2.4 Página na forma de Janela

Qualquer página criada com o Adianti Framework, seja para exibir um conteúdo HTML, um formulário, listagem, dentre outros, também pode ser exibida na forma de janela. Para tal, no lugar de estendermos a classe `TPage`, vamos estender a classe `TWindow`. Nesse caso, podemos usar métodos como `setTitle()`, para definir o título da janela, e `setSize()`, para definir seu tamanho. Todo o conteúdo adicionado à página pelo método `parent::add()` automaticamente é exibido dentro da janela.

app/control/Presentation/Pages/ExternalSystemWindowView.class.php

```
class ExternalSystemWindowView extends TWindow
{
    public function __construct()
    {
        parent::__construct();
        parent::setTitle(_t('PDF inside Window'));
        parent::setSize(0.8, 650);

        $iframe = new TElement('iframe');
        $iframe->id = "iframe_external";
        $iframe->src = "http://www.adianti.com.br/framework_files/template-material/";
        $iframe->frameborder = "0";
        $iframe->scrolling = "yes";
        $iframe->width = "100%";
        $iframe->height = "600px";

        parent::add($iframe);
    }
}
```

A figura a seguir demonstra a janela criada com um sistema externo embarcado.

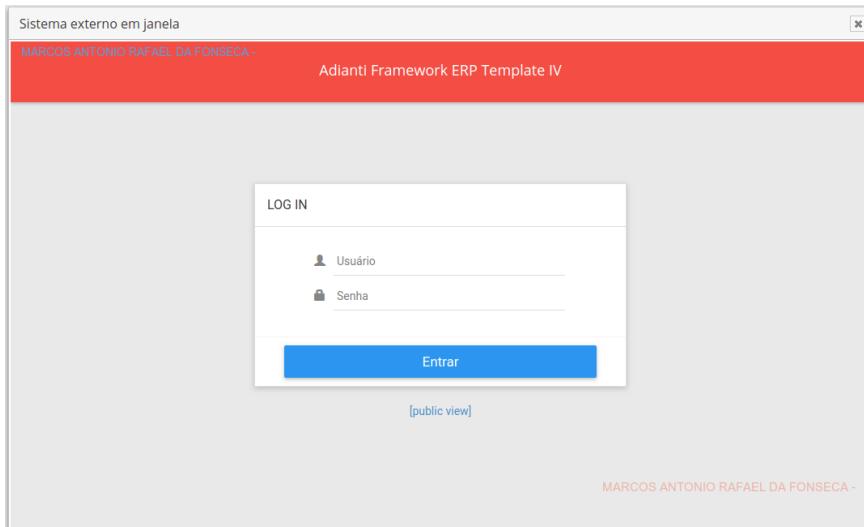


Figura 21 Janela com sistema externo

4.2.5 Abrindo janela sob demanda

No exemplo anterior, mostramos uma página do tipo Janela (`TWindow`). Mas uma janela também pode ser criada por uma página comum (`TPage`), a qualquer momento (sob demanda). Para tal, basta utilizarmos o método `TWindow::create()`. Este método cria uma janela e retorna sua instância (`$window`). Sobre este objeto, podemos adicionar conteúdo com o método `add()`. Quando tivermos adicionado o conteúdo, e quisermos exibi-lo, basta executar o método `show()`.

`app/control/Presentation/Pages/OnDemandWindowView.class.php`

```
<?php
class OnDemandWindowView extends TPage
{
    public function __construct()
    {
        parent::__construct();

        // instancia janela sob demanda
        $window = TWindow::create(_t('On demand window'), 0.8, null);

        // carrega um html
        $html = new THtmlRenderer('app/resources/page.html');

        $replaces = [];
        $replaces['title'] = 'Panel title';
        $replaces['footer'] = 'Panel footer';
        $replaces['name'] = 'Someone famous';

        // habilita seção main, passando replaces
        $html->enableSection('main', $replaces);

        $window->add($html);
        $window->show();
    }
}
```

4.2.6 Janela Modal

Nos exemplos anteriores, mostramos exemplos com Janelas contendo barra de título, e botão para realizar o fechamento da mesma. Neste próximo exemplo, vamos mostrar como construir uma janela modal.

A janela modal não terá barra de título. Além disso, terá o espaçamento reduzido entre o conteúdo e a borda da mesma. Além disso, o usuário terá desabilitado a tecla ESC para fechamento da mesma. Dessa forma, a única maneira de fechar a modal, será por meio de um botão dentro da mesma criada pelo desenvolvedor.

Para definir estas características, utilizamos os métodos `removePadding()` para reduzir o espaçamento do conteúdo para a borda; `removeTitleBar()` para remover a barra de título da janela; `disableEscape()` para desabilitar o ESC para fechamento da janela.

Dentro da janela, será exibido o template `modal.html`, que além de um conteúdo HTML, possui um botão que realiza manualmente o fechamento da janela.

app/control/Presentation/Pages/ModalWindowView.class.php

```
<?php
class ModalWindowView extends TWindow
{
    public function __construct()
    {
        parent::__construct();
        parent::setSize(0.8, null);
        parent::removePadding();
        parent::removeTitleBar();
        parent::disableEscape();

        // with: 500, height: automatic
        parent::setSize(0.6, null); // use 0.6, 0.4 (for relative sizes 60%, 40%)

        // create the HTML Renderer
        $this->html = new THtmlRenderer('app/resources/modal.html');

        $replaces = [];
        $replaces['title'] = 'Panel title';
        $replaces['footer'] = 'Panel footer';
        $replaces['name'] = 'Someone famous';

        // replace the main section variables
        $this->html->enableSection('main', $replaces);

        parent::add($this->html);
    }
}
```

A seguir, é apresentado o HTML exibido dentro da modal. Veja que existe um botão de fechamento (Close) cuja ação de click faz o fechamento manual da janela.

app/resources/modal.html

```
<!--[main]-->
<div class="card panel panel-default">
    <div class="card-header panel-heading">
        <div class="card-title panel-title" style="float:left"> {$title} </div>
        <div class="header-actions">
            <div class="btn btn-danger" style="float:right" id="close-modal"> Close </div>
        </div>
    </div>
    <div class="card-body panel-body">
        <blockquote>
            <p>Lorem ipsum dolor sit amet, consectetur adipiscing elit. ...</p>
            <footer> <b>{$name}</b> in ...</footer>
        </blockquote>
    </div>
    <div class="card-footer panel-footer"> {$footer} </div>
</div>

<script>
$('#close-modal').click( function() {
    $('[widget="TWindow"]').remove();
});
</script>

<!--[/main]-->
```

A figura a seguir demonstra a janela Modal criada.

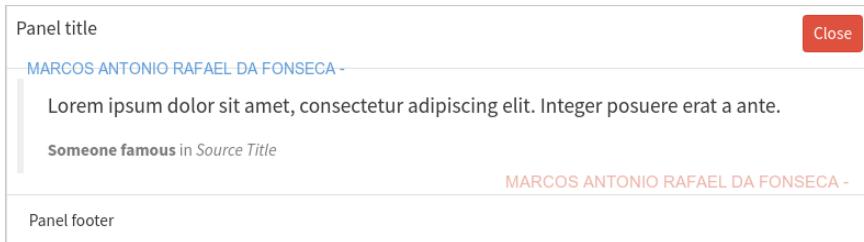


Figura 22 Janela Modal

4.2.7 Cortina lateral

Neste próximo exemplo, demonstramos como exibir um conteúdo em cortina lateral. Cortina lateral é o nome técnico dado à uma camada que se desloca lateralmente, exibindo conteúdo de uma página. Tem o nome de cortina, pois ela se desloca lateralmente em um sentido para apresentar o conteúdo, e também no sentido oposto para efetuar o recolhimento da tela, escondendo o conteúdo.

Qualquer página no Framework pode ser apresentada na forma de uma cortina lateral. Basta a utilização do método `setTargetContainer()`. Este método indica que o conteúdo da página deve ser aberto em uma `<div>` com o nome `adianti_right_panel`. O Framework faz o tratamento automático de abertura e fechamento desta cortina.

```
app/control/Presentation/Pages/SidePageView.class.php
class SidePageView extends TPage
{
    public function __construct()
    {
        parent::__construct();
        parent::setTargetContainer('adianti_right_panel');

        // create the HTML Renderer
        $this->html = new THtmlRenderer('app/resources/side.html');

        $replaces = [];
        $replaces['title'] = 'Panel title';
        $replaces['footer'] = 'Panel footer';
        $replaces['name'] = 'Someone famous';

        // replace the main section variables
        $this->html->enableSection('main', $replaces);

        parent::add($this->html);
    }

    public static function onClose($param)
    {
        TScript::create("Template.closeRightPanel()");
    }
}
```

A classe `SidePageView` exibe em tela o HTML `side.html`. Este HTML possui um botão de fechamento vinculado ao método `onClose()` da classe `SidePageView`, que por sua vez faz o fechamento da cortina pelo método `Template.closeRightPanel()`. Verja que a ação ainda informa `static=1`, uma vez que o método em questão é estático, e não requer o recarregamento da página para executar sua ação.

app/resources/side.html

```
<!--[main]-->
<div class="card panel panel-default">
    <div class="card-header panel-heading">
        <div class="card-title panel-title" style="float:left"> {$title} </div>

        <div class="header-actions">
            <a class="btn btn-default" generator="adianti"
                href="index.php?cClass=SidePageView&method=onClose&static=1"
                style="float:right">
                <i class="fa fa-times red" aria-hidden="true"></i>
                Close
            </a>
        </div>
    </div>
    <div class="card-body panel-body">
        <blockquote>
            <p>Lorem ipsum dolor sit amet...</p>
            <footer> <b>{$name}</b> in ...</footer>
        </blockquote>
    </div>
    <div class="card-footer panel-footer"> {$footer} </div>
</div>
<!--[/main]-->
```

A figura a seguir demonstra a cortina lateral.

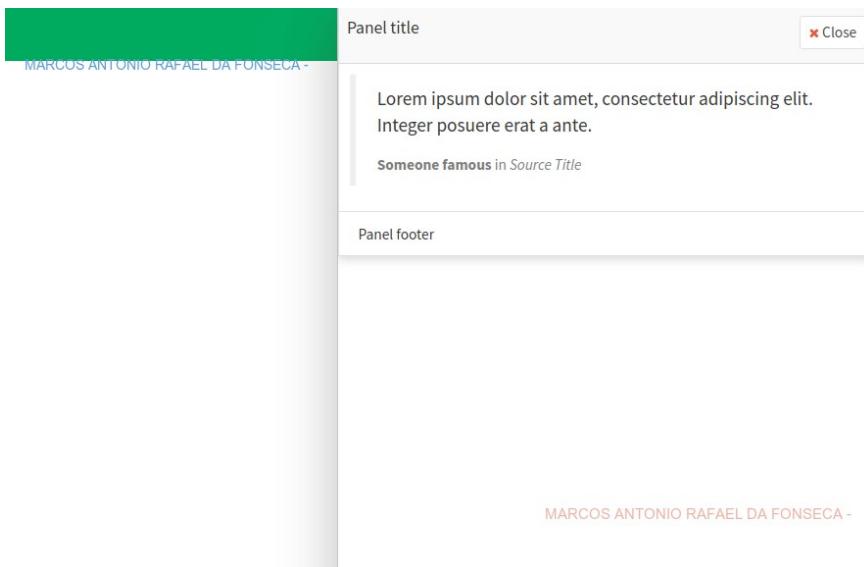


Figura 23 Cortina lateral

4.3 Containers

Um Container é um objeto que pode conter e organizar outros objetos. No Adianti Framework existem vários containers, que podem ser utilizados em conjunto. Na prática isto significa que podemos não somente colocar objetos dentro de containers, mas também containers dentro de containers, criando interfaces elaboradas.

4.3.1 Tabela

A tabela é o container mais utilizado pois permite organizar o visual dos componentes da tela em linhas e colunas. O componente que permite criar um “visual” de tabelas no Adianti Framework chama-se **TTable**. Podemos ter várias tabelas (**TTable**) dentro de uma mesma página. O componente **TTable** possui um método chamado **addRow()**, que permite adicionar uma linha na tabela. Este método retorna um objeto **TTableRow** e este, possui um método chamado **addCell()**, que adiciona uma célula dentro da linha. Cada célula representa uma coluna na tabela. O método **addCell()**, retorna um objeto da classe **TTableCell**. Sobre este objeto podemos alterar o atributo **->colspan**, quando é necessário mesclar duas células.

No exemplo que será exibido em seguida, criaremos uma tabela (**TTable**), que conterá outras duas tabelas em seu interior. Para tal, criamos inicialmente a tabela principal (**\$table**) e um objeto de título (**\$title**). Este objeto de título adicionamos na primeira linha da tabela. Logo em seguida, criamos mais duas sub-tabelas (**\$table1** e **\$table2**). Cada uma destas tabelas conterá duas linhas com alguns objetos em seu interior como **TLabel** (rótulo de texto) e **TEntry** (caixa de texto). Ao final, cada uma das duas sub-tabelas (**\$table1** e **\$table2**) são acrescentadas à tabela principal. O programa termina quando acrescentamos a tabela principal (**\$table**) à página.

Obs: Sobre vários objetos do framework como uma célula (**TTableCell**), ou tabela (**TTable**) podemos definir atributos de renderização (DOM) diretamente, como “**->width**”.

app/control/Presentation/Containers/ContainerTableView.class.php

```
<?php
class ContainerTableView extends TPage
{
    function __construct()
    {
        parent::__construct();

        // cria a tabela principal
        $table = new TTable;

        // cria um label
        $title = new TLabel('Table Layout');
        $title->setFontSize(18);
        $title->setFontFace('Arial');
        $title->setFontColor('red');

        // adiciona uma linha e uma célula
        $title = $table->addRow()->addCell($title);
```

```

// mescla duas células
$title->colspan = 2;

// cria duas outras tabelas
$stable1 = new TTable;
$stable2 = new TTable;

// cria vários campos
$id      = new TEntry('id');
$name    = new TEntry('name');
$address = new TEntry('address');
$telephone = new TEntry('telephone');
$label1  = new TLabel('Code');
$label2  = new TLabel('Name');
$label4  = new TLabel('Address');
$label5  = new TLabel('Telephone');

$row = $stable1->addRow();
$row->addCell($label1);
$row->addCell($id);

$row = $stable1->addRow();
$row->addCell($label2);
$row->addCell($name);

$row = $stable2->addRow();
$row->addCell($label4);
$row->addCell($address);

$row = $stable2->addRow();
$row->addCell($label5);
$row->addCell($telephone);

// acrescenta as sub-tabelas na tabela principal.
$row=$stable->addRow();
$row->addCell($stable1);
$row->addCell($stable2);

parent::add($stable);
}

}

```

Obs: Este código é uma versão simplificado do mesmo que se encontra no tutor.

Na figura a seguir, podemos conferir o resultado deste programa.

Table Layout

Code	MARCOS ANTONIO RAFAEL DA FONSECA	Address	
Name		Telephone	MARCOS ANTONIO RAFAEL DA FONSECA -

Figura 24 Tabelas

4.3.2 Lidando com colunas em tabelas

No exemplo anterior, vimos como manipular tabelas por meio de métodos como `addRow()`, que adiciona uma linha na tabela e `addCell()`, que adiciona uma célula na linha. Estes métodos nos permitem construir praticamente qualquer tipo de tabela, mas em alguns casos, seu uso pode parecer oneroso, em virtude de termos de realizar várias chamadas de métodos.

Pensando em simplificar a construção de tabelas, alguns métodos foram criados, dentre eles o método `addRowSet()`. O objetivo do método `addRowSet()` é adicionar uma linha e já informar os vários conteúdos que formarão aquela linha, de maneira simples e rápida. O método `addRowSet()` recebe N parâmetros, onde cada parâmetro corresponde a uma coluna da tabela.

Neste exemplo, está sendo construída uma tabela com um título que ocupa duas colunas (`colspan=2`). Em seguida, são declarados alguns objetos (`TEntry`, `TCombo`, etc). Por fim, são adicionadas várias linhas na tabela, cada qual contendo um label (`TLabel`), e um objeto, formando duas colunas.

app/control/Presentation/Containers/ContainerTableColumnsView.class.php

```
<?php
class ContainerTableColumnsView extends TPage
{
    function __construct()
    {
        parent::__construct();

        // cria a tabela
        $table = new TTable;

        // cria um label para o título
        $title = new TLabel('Table Columns');
        $title->setFontSize(18);
        $title->setFontFace('Arial');
        $title->setFontColor('red');

        // adiciona a linha para o título
        $row = $table->addRow();
        $title = $row->addCell($title);
        $title->colspan = 2;

        // cria uma série de objetos
        $id      = new TEntry('id');
        $name   = new TEntry('name');
        $address = new TEntry('address');
        $telephone = new TEntry('telephone');
        $city   = new TCombo('city');
        $text   = new TText('text');

        $items   = array();
        $items['1'] = 'Porto Alegre';
        $items['2'] = 'Lajeado';
        $city->addItems($items);
        $id->setSize(70);
```

```

    // adiciona linhas para os objetos
    $table->addRowSet( new TLabel('Code'),      $id );
    $table->addRowSet( new TLabel('Name'),       $name );
    $table->addRowSet( new TLabel('City'),        $city );
    $table->addRowSet( new TLabel('Address'),     $address );
    $table->addRowSet( new TLabel('Telephone'),   $telephone );

    parent::add($table);
}
}

```

Na figura a seguir, podemos conferir o resultado desse programa.

Table Columns	
Code	MARCOS ANTONIO RAFAEL DA FONSECA -
Name	
City	▼
Address	
Telephone	MARCOS ANTONIO RAFAEL DA FONSECA -

Figura 25 Tabelas com colunas

4.3.3 Trabalhando com células mescladas

No exemplo anterior, foi visto que o método `addRowSet()` facilita em muito a criação de colunas em tabelas. No primeiro exemplo sobre tabelas, foi visto também que podemos definir o atributo `colspan` (`$celula->colspan = 2`) de uma célula, para fazer com que ela ocupe mais de uma coluna, mesclando-a.

Para tornar ainda mais rápida a criação de tabelas com conteúdos mesclados, algumas facilidades foram criadas e podem ser vistas neste exemplo. O objetivo deste exemplo é criar uma tabela comum com duas colunas (label e objeto). Entretanto alguns elementos da segunda coluna, são formados por mais de um objeto, como a coluna Value, que possui o valor mínimo e máximo, e a coluna Date, que possui os objetos: data inicial e final. Mesmo sendo formados por mais de um elemento de entrada de dados, estes objetos devem ficar alinhados com os demais elementos da tabela.

Para criar este visual, podemos recorrer a duas abordagens. No primeiro caso (Value), podemos simplesmente chamar o método `addRowSet()`, e na segunda posição passar um vetor (array) com os vários elementos que formam a segunda coluna. Assim, eles ocuparão todos o espaço de apenas um objeto.

A segunda abordagem é utilizar o método `TTableRow::addMultiCell()`. Este método adiciona uma célula formada por vários elementos dentro de uma linha. Estas duas abordagens geram absolutamente o mesmo resultado, cabendo ao desenvolvedor escolher a que seja mais adequada.

app/control/Presentation/Containers/ContainerTableMultiCellView.class.php

```
<?php
class ContainerTableMultiCellView extends TPage
{
    function __construct()
    {
        parent::__construct();
        // cria a tabela
        $table = new TTable;

        // cria um label para o título
        $title = new TLabel('Table Multi Cell');
        $title->setFontSize(18);
        $title->setFontFace('Arial');
        $title->setFontColor('red');

        // adiciona um alinha para o título
        $row = $table->addRow();
        $title = $row->addCell($title);
        $title->colspan = 2;

        // cria uma série de campos
        $id      = new TEntry('id');
        $name    = new TEntry('name');
        $min     = new TEntry('min');
        $max     = new TEntry('max');
        $start_date = new TDate('start_date');
        $end_date = new TDate('end_date');
        $address = new TEntry('address');
        // ajustes de tamanho
        $id->setSize(70);
        $start_date->setSize(70);
        $end_date->setSize(70);
        $min->setSize(87);
        $max->setSize(87);

        // adiciona linhas para os campos
        $table->addRowSet( new TLabel('Code'), $id );
        $table->addRowSet( new TLabel('Name'), $name );

        // primeira abordagem: vetor para reunir os campos
        $table->addRowSet( new TLabel('Value'), array( $min, new TLabel('To'), $max ) );

        // segunda abordagem: método addMultiCell()
        $row = $table->addRow();
        $row->addCell(new TLabel('Date'));
        $row->addMultiCell( $start_date, $end_date );

        $table->addRowSet( new TLabel('Address'), $address );
        parent::add($table);
    }
}
```

Na figura a seguir, podemos conferir o resultado desse programa.

Table Multi Cell

Code	MARCOS ANTONIO RAFAEL DA FONSECA -	
Name	<input type="text"/>	
Value	<input type="text"/>	To <input type="text"/>
Date	<input type="text"/>	<input type="button" value="Calendar"/>
Address	<input type="text"/> MARCOS ANTONIO RAFAEL DA FONSECA -	

Figura 26 Tabela com células mescladas

4.3.4 Painel

O componente **TTable** é o container mais utilizado na maioria das vezes, uma vez que a maioria dos formulários da aplicação segue um padrão visual de linhas e colunas. Entretanto, às vezes precisamos fugir deste rigor linear. Para dar maior liberdade na disposição de componentes de apresentação, foi criado o container **TPanel**. Este container permite dispor componentes em coordenadas absolutas dentro de sua área, sendo que essas coordenadas são expressas em pixels como coordenadas X e Y.

Neste exemplo, criamos um painel de 480 pixels de largura por 260 pixels de altura. Após, são criados vários outros objetos como **TLabel**, para exibir um rótulo de texto e **TImage** para exibir uma imagem. Note que o objeto **TLabel** possui vários métodos para definir as características do texto como: **setFontSize()**, para definir o tamanho da fonte; **setFontFace()**, para definir a fonte; e **setFontColor()**, para definir a cor. Após, ainda são criados outros componentes de caixa de texto (**TEntry**) e também uma combo (**TCombo**). Estes componentes possuem o método **setSize()** para determinar o seu tamanho em pixels na tela.

Após instanciarmos vários componentes, é momento de posicionarmos estes objetos dentro do painel. Esta operação é realizada pelo método **put()**, que recebe como parâmetros: o próprio componente, a coluna (em pixels), e a linha (em pixels).

app/control/Presentation/Containers/ContainerPanelView.class.php

```
<?php
class ContainerPanelView extends TPage
{
    function __construct()
    {
        parent::__construct();
        $panel = new TPanel(480, 260); // cria o painel principal

        $titulo = new TLabel('Panel Layout');
        $titulo->setFontSize(18);
        $titulo->setFontFace('Arial');
        $titulo->setFontColor('red');
        $panel->put($titulo, 120, 4);

        $imagem = new TImage('app/images/mouse.png');
        $panel->put($imagem, 260, 140);
    }
}
```

```

$id      = new TEntry('id');
$name    = new TEntry('name');
...
$id->setSize(70);
$name->setSize(140);
...
// cria os rótulos de texto
$label1=new TLabel('Code');
$label2=new TLabel('Name');
...
// posiciona os componentes
$panel->put($label1,    10,  40);
$panel->put($id,        10,  60);
$panel->put($label2,    30,  90);
$panel->put($name,      40, 110);
...
parent::add($panel);
}
}

```

Na próxima figura, podemos ver o resultado da execução deste programa.



Figura 27 Painéis

4.3.5 Notebook

Por mais versáteis que sejam os containers `TTable` e `TPanel`, às vezes estes não são suficientes para comportar a quantidade de componentes que precisamos apresentar em tela. Nestes casos, é necessário recorrermos a containers que nos ofereçam maiores possibilidades quanto a ocupação de espaços. Um desses containers é o notebook. Um notebook divide um espaço em abas, exibindo apenas uma aba de cada vez.

No Adianti Framework, um notebook é representado pelo container `TNotebook`. Um notebook possui o método `appendPage()`, responsável por acrescentar uma aba ao notebook, que recebe como primeiro parâmetro o título da aba e como segundo parâmetro um objeto representando o conteúdo da aba.

Neste exemplo, criamos uma página contendo um notebook com duas abas. A primeira aba contém o objeto `$table1`, que é uma tabela com várias linhas com objetos como `TLabel` (rótulos de texto) e `TEntry` (caixa de texto). Já a segunda aba contém também outra tabela (`$table2`), também com algumas linhas com campos.

app/control/Presentation/Containers/ContainerNotebookView.class.php

```
<?php
class ContainerNotebookView extends TPage
{
    function __construct()
    {
        parent::__construct();
        $notebook = new TNotebook; // cria o notebook

        $table1 = new TTable;
        $table2 = new TTable;

        $notebook->appendPage('Dados básicos', $table1); // primeira aba
        $notebook->appendPage('Outros dados', $table2); // segunda aba

        $row = $table1->addRow();
        $row->addCell(new TLabel('Field1:'));
        $row->addCell(new TEntry('field1'));

        $row = $table1->addRow();
        $row->addCell(new TLabel('Field2:'));
        $row->addCell(new TEntry('field2'));

        $row = $table1->addRow();
        $row->addCell(new TLabel('Field3:'));
        $row->addCell(new TEntry('field3'));

        $row = $table2->addRow();
        $row->addCell(new TLabel('Field4:'));
        $row->addCell(new TEntry('field4'));

        $row = $table2->addRow();
        $row->addCell(new TLabel('Field5:'));
        $row->addCell(new TEntry('field5'));

        parent::add($notebook); // adiciona o notebook na página
    }
}
```

Na próxima figura, podemos ver o resultado da execução deste programa.

Figura 28 Notebook

4.3.6 Notebook Bootstrap

Você deve ter percebido que o componente **TNotebook** possui um visual peculiar no Adianti Framework. Porém, é possível modificá-lo de duas maneiras: a primeira é por meio de CSS, uma vez que a estrutura do componente é montada por HTML e CSS; a segunda maneira é utilizando um Decorator. Mas o que é um Decorator? Um Decorator é um padrão de projeto que consiste em escrever uma classe que "adiciona" funcionalidades, recursos, ou comportamento à outra já existente em tempo de execução. Isso significa que essa "agregação" de funcionalidades ocorre de maneira

dinâmica. Um Decorator diminui a necessidade de alterarmos a classe original, e também a necessidade de estendê-la, criando uma classe derivada.

O Adianti Framework possui decorators que são utilizados para “modificar” os componentes nativos e deixá-los com uma aparência diferente. No exemplo a seguir, utilizamos o `BootstrapNotebookWrapper`, que é um decorator que modifica um notebook nativo do framework e transforma-o em um notebook com a aparência e estrutura da biblioteca Bootstrap. Basta instanciarmos a classe `BootstrapNotebookWrapper` e passarmos instância de `TNotebook` como parâmetro. O restante do código-fonte foi suprimido por ser igual ao exemplo anterior.

app/control/Presentation/Containers/ContainerNotebookBootstrapView.class.php

```
<?php
class ContainerNotebookBootstrapView extends TPage
{
    function __construct()
    {
        parent::__construct();

        // cria o notebook, usando o Wrapper
        $notebook = new BootstrapNotebookWrapper( new TNotebook );

        ...
        // adiciona as abas ao notebook
        $notebook->appendPage('Dados básicos', $page1);
        $notebook->appendPage('Outra página', $page2);
        ...
    }
}
```

Na próxima figura, podemos ver o resultado da execução deste programa.



Figura 29 Notebook com Bootstrap

4.3.7 Panel group

O container `TPanelGroup` permite criarmos um painel Bootstrap nativo, que possui uma área de título (definido no método construtor), um conteúdo (definido pelo método `add()`), e uma área de rodapé (definida pelo método `addFooter()`).

```
app/control/Presentation/Containers/ContainerPanelGroupView.class.php
class ContainerPanelGroupView extends TPage
{
    function __construct()
    {
        parent::__construct();

        $panel = new TPanelGroup('Panel group title');

        $table = new TTable;
        $table->border = 1;
        $table->style = 'border-collapse:collapse';
        $table->width = '100%';
        $table->addRowSet('a1','a2');
        $table->addRowSet('b1','b2');
        $panel->add($table);
        $panel->addFooter('Panel group footer');
        parent::add($panel);
    }
}
```

Na próxima figura, podemos ver o resultado da execução deste programa.

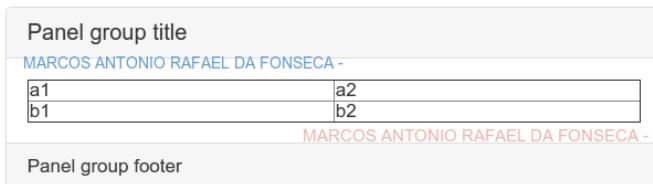


Figura 30 Panel group

4.3.8 Scroll

Normalmente quando construímos interfaces, sabemos antecipadamente a quantidade de campos a colocar na tela. Entretanto, às vezes a criação da interface se dá de maneira dinâmica e não temos como antecipar quantos campos serão colocados. Nestes casos, precisamos de componentes com barras de rolagem (*scrolling*). O Adianti Framework possui o componente **TScroll**, que se trata de uma área com barras de rolagem, podendo comportar mais componentes que a sua área visual inicial.

O componente **TScroll** possui o método **setSize()**, responsável por determinar o tamanho da área de rolagem e também o método **add()**, responsável por adicionar um conteúdo dentro da área de rolagem. Podemos adicionar ao scroll qualquer container, como uma tabela ou painel por meio do método **add()**.

No próximo exemplo, construímos uma página contendo uma área de rolagem (**\$scroll**). Determinamos o tamanho da área de rolagem com 400 pixels de largura e 200 pixels de altura por meio do método **setSize()**. Em seguida, criamos uma tabela (**new TTable**) que será acrescentada à área de rolagem por meio do método **add()**. Após, o programa entra em um FOR com 20 iterações, sendo que cada iteração cria um objeto de input (**TEntry**) e adiciona uma linha na tabela. Esses 20 elementos criados já são suficientes para percebermos o efeito das barras de rolagem.

app/control/Presentation/Containers/ContainerScrollView.class.php

```
<?php
class ContainerScrollView extends TPage
{
    function __construct()
    {
        parent::__construct();

        // cria o scroll
        $scroll = new TScroll;
        $scroll->setSize(400, 200);

        // cria uma tabela para os campos
        $fields_table = new TTable;

        // adiciona a tabela no scroll
        $scroll->add($fields_table);

        // cria uma série de campos
        for ($n=1; $n<=20; $n++)
        {
            $object = new TEntry('field' . $n);

            // adiciona uma linha para cada campo
            $row=$fields_table->addRow();
            $row->addCell(new TLabel('Field:' . $n));
            $row->addCell($object);
        }

        // adiciona o scroll na página
        parent::add($scroll);
    }
}
```

Na próxima figura, podemos ver o resultado da execução deste programa.

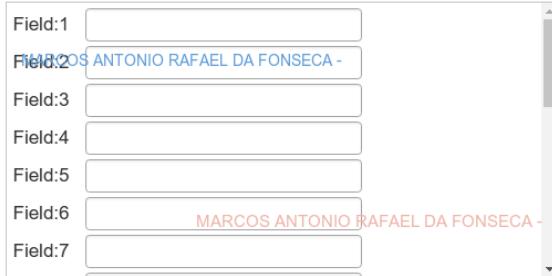


Figura 31 Scroll

4.3.9 Frame

Frequentemente quando construímos formulários, é necessário agrupar visualmente alguns elementos, por se tratarem do mesmo assunto. Para prover esta funcionalidade, o Adianti Framework possui o container moldura (**TFrame**). Um **TFrame** oferece uma área delimitada na qual podemos inserir outro container. Um **TFrame** possui um título no canto superior esquerdo.

No exemplo a seguir, criamos uma página contendo um notebook (**TNotebook**) em seu interior. Ao notebook, é acrescentada uma aba, por meio do método **appendPage()**. Dentro desta aba, é colocada uma tabela (**\$table**).

Em seguida, vários campos são criados (**\$field...**). Na tabela principal (**\$table**), são adicionadas duas linhas contendo alguns campos (**\$field1** e **\$field2**) e na terceira linha é acrescentado um frame (**new TFrame**).

Dentro do frame é acrescentada outra tabela (**\$table2**) por meio de seu método **add()**. Além disso, o método **setLegend()** é utilizado para determinar o título da moldura. O título é posicionado no canto superior esquerdo da moldura. Na tabela que está no interior da moldura (**\$table2**) são acrescentadas algumas linhas contendo campos (**\$field5**, **\$field6**) e ao final, o notebook é acrescentado à página por meio do método **parent::add()**.

app/control/Presentation/Containers/ContainerFrameView.class.php

```
<?php
class ContainerFrameView extends TPage
{
    function __construct()
    {
        parent::__construct();
        $notebook = new TNotebook;

        $table = new TTable; // cria a tabela principal
        // acrescenta uma aba
        $notebook->appendPage('Register data', $table);

        $field1 = new TEntry('field1');
        $field2 = new TEntry('field2');
        $field3 = new TEntry('field3');
        $field4 = new TEntry('field4');
        ...
        $field1->setEditable(FALSE);
        $field1->setSize(100);
        $field2->setSize(300);
        $field4->setSize(100);
        ...
        $row=$table->addRow();
        $row->addCell(new TLabel('Id:'));
        $cell = $row->addCell($field1);

        $row=$table->addRow();
        $row->addCell(new TLabel('Description:'));
        $cell=$row->addCell($field2);

        // cria o frame
        $frame = new TFrame;
        $frame->setLegend('Measures'); // define o título do frame

        $row=$table->addRow();
        $cell=$row->addCell($frame);
        $cell->colspan=2;

        $table2 = new TTable;
        $frame->add($table2); // acrescenta conteúdo ao frame
    }
}
```

```

$row=$table2->addRow();
$row->addCell(new TLabel('Stock:'));
$row->addCell($field3);
$row->addCell(new TLabel('Unit:'));
$row->addCell($field4);

$row=$table2->addRow();
$row->addCell(new TLabel('Cost Price:'));
$row->addCell($field5);
$row->addCell(new TLabel('Sell Price:'));
$row->addCell($field6);

$row=$table2->addRow();
$row->addCell(new TLabel('Net Weight:'));
$row->addCell($field7);
$row->addCell(new TLabel('Gross Weight:'));
$row->addCell($field8);

parent::add($notebook);
}

}

```

Na próxima figura, podemos ver o resultado da execução deste programa.

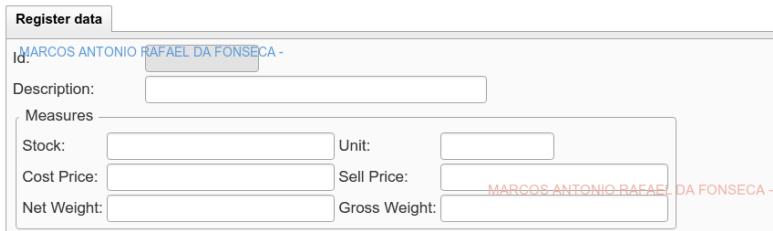


Figura 32 Frame

4.3.10 Caixas horizontais e verticais

O container **THBox** permite dispor elementos um ao lado do outro por meio de uma caixa horizontal. Já a **TVBox** permite dispor elementos um acima do outro. Neste exemplo, são criados dois notebooks (**TNotebook**), e são adicionadas duas páginas (**appendPage()**) em cada um. Por fim, está sendo usada uma caixa horizontal (**THBox**) para dispor um ao lado do outro. O método **add()** permite adicionarmos objetos a uma caixa horizontal.

app/control/Presentation/Containers/ContainerHBoxView.class.php

```

<?php
class ContainerHBoxView extends TPage
{
    function __construct()
    {
        parent::__construct();
        $notebook1 = new TNotebook;
        $notebook2 = new TNotebook;
        $notebook1->appendPage('page1', new TLabel('Page 1'));
        $notebook1->appendPage('page2', new TLabel('Page 2'));
        $notebook2->appendPage('page1', new TLabel('Page 1'));
        $notebook2->appendPage('page2', new TLabel('Page 2'));
        $notebook1->setSize(200,100);
        $notebook2->setSize(200,100);
    }
}

```

```
// cria uma caixa horizontal
$bbox = new THBox;
$bbox->add($notebook1);
$bbox->add($notebook2);

parent::add($bbox);
}

}
```

Na próxima figura, podemos ver o resultado da execução deste programa.

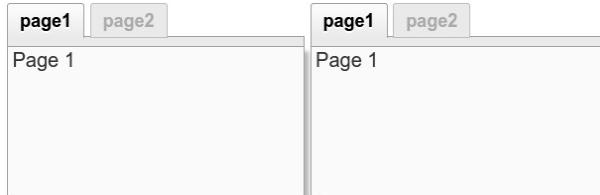


Figura 33 Caixas horizontais com notebooks

4.4 Diálogos

Em diversos momentos em uma aplicação, precisamos interagir com o usuário para exibir mensagens de êxito no cumprimento de tarefas, mensagens de falha ou mesmo indagá-lo sobre o caminho a seguir dentro da lógica de negócios. Interagimos com o usuário por meio de diálogos que podem ser: informação, erro, aviso, questionamento, e input.

4.4.1 Informação

Precisamos seguidamente informar o usuário sobre o sucesso na conclusão de tarefas: quando salvamos um registro com sucesso, quando excluímos um registro, quando o sistema finaliza um processamento. Nestes casos, utilizamos um diálogo de sucesso. No Adianti Framework, a classe `TMessage` é responsável por exibir mensagens ao usuário, sejam de sucesso ou de falhas. Nesse primeiro exemplo, vemos como exibir uma mensagem de sucesso ao usuário.

Para exibir uma mensagem de sucesso, basta instanciarmos um objeto `TMessage`. O primeiro parâmetro identifica o tipo da mensagem e deve ser '`info`' para mensagens informativas de sucesso. O segundo parâmetro informa a mensagem.

app/control/Presentation/Dialogs/DialogInformationView.class.php

```
<?php
class DialogInformationView extends TPage
{
    public function __construct()
    {
        parent::__construct();
        new TMessage('info', 'Mensagem informativa');
    }
}
```

Na próxima figura, podemos ver o resultado da execução deste programa.



Figura 34 Mensagem de informação

4.4.2 Atenção

Outro tipo de mensagem comum a ser apresentada ao usuário é a mensagem de atenção. No Adianti Framework a mesma classe utilizada para exibir mensagens informativas (**TMessage**) é utilizada para exibir mensagens de atenção. A única modificação que precisamos fazer é alterar o primeiro parâmetro '**warning**'. Este parâmetro causará a mudança no tipo de diálogo a ser exibido.

Obs: Ao criar uma mensagem pode ser passado um terceiro parâmetro que indica a ação (objeto **TAction**) a ser executada quando o usuário clicar no OK do diálogo.

[app/control/Presentation/Dialogs/DialogWarningView.class.php](#)

```
<?php
class DialogWarningView extends TPage
{
    public function __construct()
    {
        parent::__construct();
        new TMessage('warning', 'Mensagem de atenção');
    }
}
```

Na próxima figura, podemos ver o resultado da execução deste programa.



Figura 35 Mensagem de atenção

4.4.3 Erro

Outro tipo de mensagem comum a ser apresentada ao usuário é a mensagem de erros. No Adianti Framework a mesma classe utilizada para exibir mensagens informativas (**TMessage**) é utilizada para exibir mensagens de erro. A única modificação que precisamos fazer é alterar o primeiro parâmetro 'error'. Este parâmetro causará a mudança no tipo de diálogo a ser exibido.

[app/control/Presentation/Dialogs/DialogErrorView.class.php](#)

```
<?php
class DialogErrorView extends TPage
{
    public function __construct()
    {
        parent::__construct();
        new TMessage('error', 'Mensagem de erro');
    }
}
```

Na próxima figura, podemos ver o resultado da execução deste programa.



Figura 36 Mensagem de erro

4.4.4 Questionamento

Os diálogos vistos até agora (informação, atenção e erro) têm como objetivo somente apresentar o resultado de uma ação ao usuário. Mas em muitas ocasiões, precisamos não somente apresentar um resultado, mas questionar o usuário sobre a ação a ser desempenhada. Nesses casos, podemos usar o diálogo de questionamento.

No Adianti Framework, um diálogo de questionamento é realizado pelo componente **TQuestion**. Esse componente permite realizar uma pergunta ao usuário e definir duas possíveis ações: uma para resposta afirmativa (Sim) e outra para a resposta negativa (Não).

Ao trabalharmos com diálogos de questionamento, teremos o primeiro contato com o conceito de ações (*actions*). Uma ação é um objeto **TAction** que identifica um método de alguma classe que será executado. Além de identificar o método, a ação pode carregar consigo parâmetros que são passados para o método. Neste exemplo, temos duas ações (objetos **TAction**). A primeira ação (**\$action1**) será executada sempre que o usuário responder “Sim” ao questionamento realizado, e a segunda ação (**\$action2**), será executada quando o usuário responder “Não”. Ao instanciar uma ação (**new TAction**),

informamos um vetor contendo duas posições: um objeto e um método, sendo que o método deve pertencer àquele objeto. Neste caso, os métodos a serem executados são: `onAction1()` e `onAction2()`, desta mesma classe (`$this`).

O diálogo de questionamento é criado quando instanciamos o objeto `TQuestion`. Neste momento, devemos informar a mensagem do diálogo e também dois objetos `TAction`: um a ser executado caso a resposta for “Sim” e outro para a resposta “Não”. Além de executar os métodos, desejamos passar parâmetros para eles. Para passar parâmetros para os métodos das ações, podemos executar o método `setParameter()`, da classe `TAction`. Este método define um nome e um valor para o parâmetro.

Caso o usuário responda “Sim”, o método `onAction1()` é executado e uma mensagem é exibida com o valor do parâmetro. Caso contrário, o método `onAction2()` é executado, e da mesma forma, uma mensagem com o parâmetro é exibida.

`app/control/Presentation/Dialogs/DialogQuestionView.class.php`

```
<?php
class DialogQuestionView extends TPage
{
    public function __construct()
    {
        parent::__construct();

        // cria duas ações
        $action1 = new TAction(array($this, 'onAction1'));
        $action2 = new TAction(array($this, 'onAction2'));

        // define os parâmetros de cada ação
        $action1->setParameter('parameter', 1);
        $action2->setParameter('parameter', 2);

        // exibe o diálogo de questionamento
        new TQuestion('Deseja realizar esta operação ?', $action1, $action2);
    }

    function onAction1($param)
    {
        new TMessage('info',
                    "Primeira opção. Valor do parâmetro: {$param['parameter']}");
    }

    function onAction2($param)
    {
        new TMessage('info',
                    "Segunda opção. Valor do parâmetro: {$param['parameter']}");
    }
}
```

Obs: A segunda ação, que representa a resposta “Não”, é opcional. Caso ela não for informada, a opção “Não” simplesmente fecha o diálogo de questionamento.

Na próxima figura, podemos ver o resultado da execução deste programa.

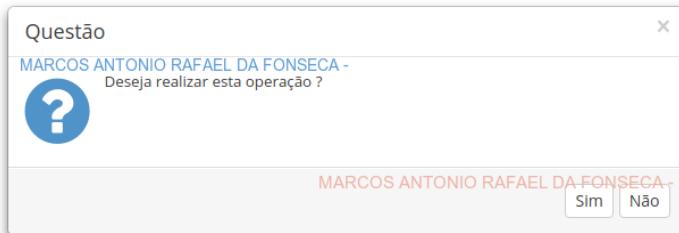


Figura 37 Diálogo de questionamento

4.4.5 Toast

Uma forma alternativa e menos invasiva que os diálogos são os Toasts. Os Toasts tem a vantagem de não necessitarem de uma ação do usuário para saírem da tela, visto que eles deixam a tela automaticamente depois de alguns instantes. Toasts podem ser utilizados para mensagens que não demandem a atenção ou ação imediata do usuário.

app/control/Presentation/Dialogs/DialogToastView.class.php

```
<?php
class DialogToastView extends TPage
{
    public function __construct()
    {
        parent::__construct();

        TToast::show('show', 'Toast test 1', 'top right', 'fa:check-circle-o' );
        TToast::show('info', 'Toast test 2', 'top right', 'fa:check-circle-o' );
        TToast::show('warning', 'Toast test 3', 'top right', 'fa:check-circle-o' );
        TToast::show('error', 'Toast test 4', 'top right', 'fa:check-circle-o' );
        TToast::show('success', 'Toast test 4', 'top right', 'fa:check-circle-o' );

        parent::add(new TXMLBreadCrumb('menu.xml', __CLASS__));
    }
}
```

A próxima figura, demonstra o Toast em ação.



Figura 38 Toast

4.4.6 Input

Além dos diálogos de apresentação de informação (**TMessage**) e de questionamento (**TQuestion**), o Adianti Framework possui também o diálogo de input (**TInputDialog**). O diálogo de input permite abrir um formulário rápido, sobre a tela do usuário, para solicitar o preenchimento de valores. Além de solicitar valores, ações podem ser programadas sobre estes valores. Diálogos de input podem ser executados a partir de simples botões, ações de datagrids, ou a qualquer momento dentro de um processo.

Para utilizar um diálogo de input, é necessário utilizar um formulário. Nesse exemplo, criaremos um formulário utilizando a classe **BootstrapFormBuilder**, que será explicada em maiores detalhes nas próximas seções do livro. A classe **BootstrapFormBuilder**, permite criar um formulário, com alguns componentes de entrada de dados, tais como input de texto, seleção de datas, dentre outros.

Nesse exemplo, no método construtor estamos construindo um diálogo de input. Para tal, estamos criando um formulário (**BootstrapFormBuilder**) com dois campos: *login* e *password*, sendo que utilizamos o componente **TEntry** para o login e **TPassword** para a senha. O método **addFields()** adiciona os campos ao formulário. Já o método **addAction()** adiciona um botão de ação ao formulário, que ao ser clicado, executará o método **onConfirm()**.

Para criar o diálogo, basta utilizar a classe **TInputDialog**, passando como parâmetros: o título do diálogo, e o formulário que será embutido no diálogo. A classe **TInputDialog** incorporará o formulário dentro do diálogo e solicitará o preenchimento ao usuário. Assim que clicar no botão, a ação correspondente é executada. Neste exemplo, o método **onConfirm()** recebe os dados via vetor de parâmetros: **\$param**, e exibe ao usuário por meio da classe **TMessage**.

app/control/Presentation/Dialogs/DialogInputView.class.php

```
<?php
class DialogInputView extends TPage
{
    public function __construct()
    {
        parent::__construct();

        $form = new BootstrapFormBuilder('input_form');

        $login = new TEntry('login');
        $pass = new TPassword('password');

        $form->addFields( [new TLabel('Login')], [$login]);
        $form->addFields( [new TLabel('Password')], [$pass]);

        $form->addAction('Confirm 1', new TAction([$this, 'onConfirm1']), 'fa:save green');

        // exibe o diálogo de input
        new TInputDialog('Input dialog title', $form);
    }
}
```

```

/**
 * exibe os resultados
 */
public function onConfirm( $param )
{
    new TMessage('info', 'Confirm : ' . json_encode($param));
}
}

```

Na próxima figura, podemos ver o resultado da execução deste programa.



Figura 39 Diálogo de input

4.5 Formulários

Nas seções anteriores, vimos como organizar visualmente os elementos em uma interface por meio de containers e também como apresentar informações ao usuário por meio de diálogos. Nesta seção, veremos como utilizar formulários para obter e registrar a entrada de informações na aplicação por meio de componentes que acompanham o Adianti Framework. Esta seção será dividida em tópicos, onde cada tópico apresentará uma característica diferente do uso de formulários.

4.5.1 Componentes para formulários

Vários componentes podem ser utilizados dentro de formulários. A seguir, uma tabela resumida contendo o nome da classe e uma breve descrição.

Tabela 2. Componentes para uso em formulários

Componente	Descrição
TButton	Botão de ação para formulários.
TCheckGroup	Campo para seleção não exclusiva de itens.
TColor	Campo para seleção de cores.
TCombo	Campo para seleção exclusiva de itens em lista.
TDate	Campo para seleção de data.
TDATETIME	Campo para seleção de data com hora.
TDBCheckGroup	Campo para seleção não exclusiva de itens a partir do banco de dados.
TDBCombo	Campo para seleção exclusiva de itens em lista a partir do banco de dados.
TDBEntry	Campo de entrada com autocomplete a partir do banco de dados.

Componente	Descrição
TDBMultiSearch	Campo para busca de um ou vários registros relacionados de uma tabela.
TDBRadioGroup	Campo para seleção exclusiva de itens a partir do banco de dados.
TDBSelect	Campo para seleção não exclusiva de itens a partir do banco de dados.
TDBSortList	Campo para ordenação de elementos a partir do banco de dados.
TDBUniqueSearch	Campo para seleção exclusiva de itens em lista a partir do banco de dados.
TEntry	Campo de entrada de texto simples.
TFile	Campo para upload de um arquivo.
THidden	Campo escondido.
THtmlEditor	Campo para edição de HTML.
TIcon	Campo seletor de ícone.
TMultiEntry	Campo para entrada de um conjunto de palavras-chave.
TMultiFile	Campo para upload de vários arquivos.
TMultiSearch	Campo para busca de um ou vários itens a partir de um vetor.
TNumeric	Campo para entrada de valores numéricos.
TPassword	Campo para digitação de senhas.
TRadioGroup	Campo para seleção exclusiva de itens.
TSeekButton	Campo para localização de campos e retorno de informações.
TSelect	Campo para seleção não exclusiva de itens.
TSlider	Campo para seleção numérica com um seletor, que pode ser movido pelo usuário dentro de uma espécie de régua.
TSortList	Campo para ordenação de elementos.
TSpinner	Campo para entrada de informações numéricas com setas para cima e para baixo para ajuste de valor (para mais ou para menos).
TText	Campo de entrada de texto com várias linhas.
TTime	Campo para seleção de horas.
TUniqueSearch	Campo para seleção exclusiva de itens em lista.

Obs: Ao longo dos próximos exemplos, procuraremos demonstrar a utilização de cada um dos componentes do Framework.

4.5.2 Formulário manual

Neste primeiro exemplo da série de formulários, construiremos um formulário manualmente. Utilizamos o nome “manual” pois neste exemplo montaremos todo o formulário (lógica e apresentação). Já nos exemplos seguintes, aprenderemos a utilizar classes construtoras de formulários, que permitem a montagem de formulários complexos com poucas linhas de código. Mas por que aprenderemos o jeito difícil se existe um jeito fácil? Por que pode existir uma situação em que o jeito fácil não resolva. Neste caso, é importante que você saiba montar um formulário manualmente.

O Framework permite desacoplar os aspectos visuais dos aspectos lógicos de um formulário, mas o que isto significa? Significa poder criar um formulário à mão livre, empacotando seus elementos com os containers que achamos melhor.

A classe **TForm** é uma classe ancestral do Framework que é utilizada para gerenciar os dados de um formulário (definição de campos, atribuição de dados, obtenção de dados, transporte de dados). Esta classe não trata da apresentação do formulário, apenas da lógica de envio e recebimento de dados. Para a apresentação, ela necessita que utilizamos outros containers, tais como tabelas, painéis, dentre outros para organizar os campos em seu interior. Portanto, a classe **TForm** somente funciona se utilizarmos classes de apresentação juntamente a ela.

Neste exemplo, utilizaremos um notebook (**TNotebook**), bem como tabelas (**TTable**), em seu interior para organizar o layout do formulário, que terá duas abas. Como essas classes cuidam somente do visual, não da postagem dos dados, utilizaremos a classe **TForm**, para cuidar do envio e recebimento dos dados. Para adicionar elementos ao formulário, devemos empacotá-los em um container (**TTable**, **TPanel**), e então, acrescentar este container ao formulário (**TForm**).

O exemplo a seguir, inicia pela criação do formulário lógico (**TForm**), e em seguida do notebook (**TNotebook**), que terá em seu interior duas tabelas (**TTable**). O formulário lógico (**TForm**) precisa estar ao redor de todos os campos do formulário para gerenciar a postagem dos mesmos. Como os campos estarão dentro do notebook, este é adicionado ao formulário. Após, usamos o método **appendPage()** para acrescentar as tabelas dentro das abas do notebook. Em seguida, criamos alguns campos (**TEntry**, **TDate**, **TCombo**, **TText**), e definimos algumas de suas propriedades. Os campos são inseridos nas tabelas por meio do método **addRowSet()**. Veja que neste exemplo, usamos a tabela para posicionar visualmente os elementos, mas poderíamos usar qualquer outro container, desde que este container esteja dentro do formulário (**TForm**). Após adicionar os campos à tabela, criamos manualmente um botão de ação, por meio da classe **TButton**, que define o método a ser executado por meio do **setAction()**.

Por fim, ao utilizarmos esta abordagem, de construção do layout separado da lógica, precisamos executar o método **setFields()** para dizer ao formulário, quais campos ele será responsável por gerenciar na postagem.

app/control/Presentation/Forms/FormCustomView.class.php

```
<?php
class FormCustomView extends TPage
{
    private $form;

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

        // cria o formulário
        $this->form = new TForm('customform');

        // cria o notebook
        $notebook = new TNotebook;
        $this->form->add($notebook);

        // cria duas tabelas, uma para cada aba
        $table1 = new TTable;
        $table2 = new TTable;

        $table1->width = '100%';
        $table2->width = '100%';

        $table1->style = 'padding: 10px';
        $table2->style = 'padding: 10px';

        // adiciona as tabelas ao notebook
        $notebook->appendPage('Page 1', $table1);
        $notebook->appendPage('Page 2', $table2);

        // cria os campos do formulário
        $field1 = new TEntry('field1');
        $field2 = new TEntry('field2');
        $field3 = new TDate('field3');
        $field4 = new TCombo('field4');
        $field5 = new TText('field5');

        $field1->setSize('30%');
        $field2->setSize('100%');
        $field3->setSize('100%');
        $field4->setSize('100%');
        $field5->setSize('100%', 70);

        $items = [ 'a' => 'Item a', 'b' => 'Item b', 'c' => 'Item c' ];
        $field4->addItems($items);

        // adiciona as linhas para os campos
        $table1->addRowSet( new TLabel('Field 1'), $field1 );
        $table1->addRowSet( new TLabel('Field 2'), $field2 );
        $table1->addRowSet( new TLabel('Field 3'), $field3 );
        $table1->addRowSet( new TLabel('Field 4'), $field4 );

        $table2->addRowSet( new TLabel('Field 5') );
        $table2->addRowSet( $field5 );

        // cria o botão de ação
        $button = new TButton('action1');
        $button->setAction(new TAction(array($this, 'onSend'))), 'Send');
        $button->setImage('fa:check-circle-o green');

        // define quais serão os campos manipulados pelo formulário
        $this->form->setFields([$field1, $field2, $field3, $field4, $field5, $button]);
```

```

$panel = new TPanelGroup(_t('Manual form'));
$panel->add($this->form);
$panel->addFooter($button);

// empacota tudo em uma caixa vertical
$ vbox = new TVBox;
$vbox->style = 'width: 100%';
$vbox->add(new TXMLBreadCrumb('menu.xml', __CLASS__));
$vbox->add($panel);

parent::add($vbox);
}

```

Por fim, temos o método `onSend()`, que cuida do tratamento da postagem dos dados. Neste caso, simplesmente obtemos os dados pelo método `getData()`, e ao final, exibimos estes dados, codificando-os em JSON, para facilitar a apresentação.

```

public function onSend($param)
{
    $data = $this->form->getData();
    $this->form->setData($data);

    new TMessage('info', str_replace(',', '<br>', json_encode($data)));
}
}

```

Na próxima figura, podemos ver o resultado da execução deste programa.

Figura 40 Formulário manual

Obs: Use essa abordagem sempre que for necessário construir um formulário com organização visual fora do padrão estabelecido pelas classes básicas do framework.

4.5.3 Formulários Bootstrap

No exemplo anterior vimos como construir um formulário manualmente no Framework utilizando a classe `TForm` combinada com containers para organizar seu visual. Porém, na maioria das vezes, os formulários possuem um visual padronizado e portanto podemos criar uma classe que trate tanto da lógica, quanto da apresentação. Normalmente, mais de 90% dos formulários de uma aplicação possuem uma aparência padronizada. Assim, a partir deste exemplo, vamos utilizar uma classe que automatiza a montagem de formulários responsivos com a biblioteca Bootstrap.

Para a montagem do formulário, utilizaremos a classe `BootstrapFormBuilder`, instanciada logo no início do programa. Logo em seguida, criaremos uma série de objetos para entrada de dados (`TEntry`, `TDateTime`, `TColor`, `TSpinner`, `TCombo`, `TText`). No método construtor, identificamos o nome do input, que será utilizado para obter o seu conteúdo posteriormente no lado do servidor, e para a gravação dos dados.

Após a criação dos objetos, definimos suas propriedades. O método `setEditable()` liga ou desliga a edição de um campo `TEntry`; o método `setMask()` define a máscara de tela de um campo `TDate` ou `TDateTime`, já o método `setDatabaseMask()` define a máscara que o campo tem no banco de dados (o valor será convertido automaticamente para a máscara de tela na edição e para a máscara de banco na postagem); O método `setNumericMask()` habilita uma máscara de digitação numérica em um campo `TEntry`; O método `addItems()` preenche um campo `TCombo` com um vetor de opções; O método `setRange()` define o intervalo de seleção numérica (mínimo, máximo, passada) de um `TSpinner`; Já o método `setSize()` define o tamanho dos objetos em tela, e `setValue()` define um valor inicial para o campo.

A classe `BootstrapFormBuilder` trabalha com o conceito de slots de campos. Cada vez que usamos o método `addField()`, adicionamos alguns slots com objetos. Ao adicionarmos 1 slot de conteúdo, ele ocupará todo espaço disponível (`col-sm-12`). Ao adicionarmos 2 slots de conteúdo, o primeiro ocupará menos espaço que o segundo, (`col-sm-2`, `col-sm-10`), e ao adicionarmos 4 slots de conteúdo, o primeiro e o terceiro (geralmente labels) ocuparão menos espaço que o segundo e quarto (geralmente inputs) com as proporções: `col-sm-2`, `col-sm-4`, `col-sm-2`, `col-sm-4`. Cada slot de conteúdo recebe um vetor, sendo que este vetor pode ter somente uma posição (objeto), ou mais de uma, fazendo com que vários objetos compartilhem o mesmo espaço. Posteriormente, veremos como mudar a proporção dos slots manualmente.

Após adicionarmos os campos ao formulário, alteramos outras propriedades como o `placeholder` de um campo, ou a dica de preenchimento, pelo método `setTip()`. O método `addAction()` é utilizado para adicionar uma ação de envio de dados ao formulário. Este método recebe o nome do botão, método a ser executado e ícone. Já o método `addHeaderAction()` é utilizado para adicionar uma ação na barra de título. Ao final, os elementos são empacotados por uma caixa vertical (`TVBox`), e esta é adicionada à página.

app/control/Presentation/Forms/FormBuilderView.class.php

```
<?php
class FormBuilderView extends TPage
{
    private $form;
    function __construct()
    {
        parent::__construct();

        // cria o formulário
        $this->form = new BootstrapFormBuilder;
        $this->form->setFormTitle(_t('Bootstrap form'));
        $this->form->generateAria(); // gera automaticamente os aria-label

        // cria os campos do formulário
        $id          = new TEntry('id');
        $description = new TEntry('description');
        $password    = new TPassword('password');
        $created     = new TDateTime('created');
        $expires     = new TDate('expires');
        $value       = new TEntry('value');
        $color       = new TColor('color');
        $weight      = new TSpinner('weight');
        $type        = new TCombo('type');
        $text         = new TText('text');

        $id->setEditable(FALSE);
        $created->setMask('dd/mm/yyyy hh:ii');
        $expires->setMask('dd/mm/yyyy');
        $created->setDatabaseMask('yyyy-mm-dd hh:ii');
        $expires->setDatabaseMask('yyyy-mm-dd');
        $value->setNumericMask(2, ',', '.', true);
        $value->setSize('100%');
        $color->setSize('100%');
        $created->setSize('100%');
        $expires->setSize('100%');
        $weight->setRange(1,100,0.1);
        $weight->setSize('100%');
        $type->setSize('100%');
        $type->addItems( ['a' => 'Type a', 'b' => 'Type b', 'c' => 'Type c'] );

        $created->setValue( date('Y-m-d H:i') );
        $expires->setValue( date('Y-m-d', strtotime("+1 days")) );
        $value->setValue(123.45);
        $weight->setValue(30);
        $color->setValue('#FF0000');
        $type->setValue('a');

        // adiciona os campos ao formulário
        $this->form->addField( [new TLabel('Id')], [$id] );
        $this->form->addField( [new TLabel('Description')], [$description] );
        $this->form->addField( [new TLabel('Password')], [$password] );
        $this->form->addField( [new TLabel('Created at')], [$created] );
        $this->form->addField( [new TLabel('Expires at')], [$expires] );
        $this->form->addField( [new TLabel('Value')], [$value] );
        $this->form->addField( [new TLabel('Color')], [$color] );
        $this->form->addField( [new TLabel('Weight')], [$weight] );
        $this->form->addField( [new TLabel('Type')], [$type] );

        $description->placeholder = 'Description placeholder';
        $description->setTip('Tip for description');

        $label = new TLabel('Division', '#D78B6', 12, 'bi');
        $label->style='text-align:left; border-bottom:1px solid #c0c0c0; width:100%';
        $this->form->addContent( [$label] );
    }
}
```

```

$this->form->addFields( [ new TLabel('Description')], [$text] );
$text->setSize('100%', 50);

// define ação do formulário
$this->form->addAction('Send', new TAction(array($this, 'onSend')),
    'fa:check-circle-o green');
$this->form->addHeaderAction('Send', new TAction(array($this, 'onSend')),
    'fa:rocket orange');

// empacota tudo em uma caixa vertical
$ vbox = new TVBox;
$ vbox->style = 'width: 100%';
$ vbox->add(new TXMLBreadCrumb('menu.xml', __CLASS__));
$ vbox->add($this->form);

parent::add($vbox);
}

```

Quando o botão de ação for clicado, o método `onSend()` da mesma classe é executado. Por padrão, todos os métodos executados pelo Adianti Framework recebem um vetor de parâmetros (`$param`), que corresponde aos dados da requisição.

O método `onSend()` obtém os valores do formulário por meio do método `getData()`. Este método retorna um objeto contendo os dados, onde cada atributo identifica um campo do formulário. Este método recebe opcionalmente o nome de uma classe Active Record, que se informada designará o tipo do objeto retornado. O método `setData()` é usado para preencher o formulário novamente com os dados preenchidos, para que o mesmo não fique vazio após a postagem. Após obter os dados postados, simplesmente montamos uma mensagem com os valores do formulário na forma de string, e a apresentamos em tela, por meio da classe `TMessage`.

```

public function onSend($param)
{
    $data = $this->form->getData();

    // put the data back to the form
    $this->form->setData($data);

    // creates a string with the form element's values
    $message = 'Id: ' . $data->id . '<br>';
    $message.= 'Description : ' . $data->description . '<br>';
    $message.= 'Password : ' . $data->password . '<br>';
    $message.= 'Created: ' . $data->created . '<br>';
    $message.= 'Expires: ' . $data->expires . '<br>';
    $message.= 'Value : ' . $data->value . '<br>';
    $message.= 'Color : ' . $data->color . '<br>';
    $message.= 'Weight : ' . $data->weight . '<br>';
    $message.= 'Type : ' . $data->type . '<br>';
    $message.= 'Text : ' . $data->text . '<br>';

    // show the message
    new TMessage('info', $message);
}

```

Na próxima figura, podemos ver o resultado da execução deste programa.

The screenshot shows a web form titled "Formulário Bootstrap". It includes fields for "Id", "Description" (containing "desc"), "Password" (containing "****"), "Created at" (set to "30/07/2018 23:51"), "Expires at" (set to "31/07/2018"), "Value" (containing "12.345,67"), "Color" (set to "#ff0000"), "Weight" (set to "30"), and "Type" (set to "Type a"). Below these, there's a section labeled "Division" with a "Description" field containing "abc". At the bottom left is a "Send" button, and at the bottom right is a signature "MARCOS ANTONIO RAFAEL DA FONSECA".

Figura 41 Formulário Bootstrap

4.5.4 Formulário Bootstrap em colunas

No exemplo anterior, vimos que ao utilizarmos a classe `BootstrapFormBuilder`, ela organiza os campos na tela em slots com proporções pré-definidas. Neste exemplo, vamos mostrar exatamente quais são as proporções pré-definidas, além de demonstrar como configurar as proporções manualmente.

Inicialmente, vamos criar um formulário (`BootstrapFormBuilder`), e adicionar uma aba nele com o método `appendPage()`. Em seguida, vamos demonstrar como a classe `BootstrapFormBuilder` distribui automaticamente os campos no formulário. Para tal, vamos executar o método `addFields()` diversas vezes, sendo cada vez com uma quantidade diferente de slots com campos (2, 3, 4, 5, 6).

Em seguida, vamos adicionar uma nova aba com o método `appendPage()`. Nesta nova aba, também vamos adicionar linhas com slots de tamanhos variados pelo método `addFields()`. Entretanto, diferentemente da primeira vez, vamos usar uma variável chamada `$row`, para guardar o retorno do método `addFields()`. Sobre esta variável, vamos definir um atributo chamado `layout`, que receberá um vetor com as proporções dos slots da linha. Estas classes (`col-sm-2`) são da biblioteca Bootstrap.

Ao final, adicionamos um botão de ação, vinculado ao método `onSend()`. E em seguida, adicionamos o formulário em uma caixa (`TBox`), e esta, à página.

app/control/Presentation/Forms/FormBuilderGridView.class.php

```
<?php
class FormBuilderGridView extends TPage
{
    private $form;
```

```

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

    $this->form = new BootstrapFormBuilder;
    $this->form->setFormTitle(_t('Bootstrap form grid'));

    // adiciona uma aba para mostrar o ajuste automático
    $this->form->appendPage('Automatic form grid');

    // adiciona slots de tamanhos variados
    $this->form->addFields( [ new TLabel('2 slots') ],
        [ new TEntry('field_1') ] );

    $this->form->addFields( [ new TLabel('3 slots') ],
        [ new TEntry('field_2') ],
        [ new TEntry('field_3') ] );

    $this->form->addFields( [ new TLabel('4 slots') ],
        [ new TEntry('field_4') ],
        [ new TEntry('field_5') ],
        [ new TEntry('field_6') ] );

    $this->form->addFields( [ new TLabel('5 slots') ],
        [ new TEntry('field_7') ],
        [ new TEntry('field_8') ],
        [ new TEntry('field_9') ],
        [ new TEntry('field_10') ] );

    $this->form->addFields( [ new TLabel('6 slots') ],
        [ new TEntry('field_11') ],
        [ new TEntry('field_12') ],
        [ new TEntry('field_13') ],
        [ new TEntry('field_14') ],
        [ new TEntry('field_15') ] );

    // adiciona outra aba para mostrar o ajuste manual
    $this->form->appendPage('Manual form grid');

    // adiciona slots com tamanhos personalizados
    $row = $this->form->addFields( [ new TLabel('Custom 1') ],
        [ new TEntry('field_16') ],
        [ new TEntry('field_17') ],
        [ new TEntry('field_18') ] );

    $row->layout = ['col-sm-2 control-label', 'col-sm-4', 'col-sm-4', 'col-sm-2'];

    $row = $this->form->addFields( [ new TLabel('Custom 2') ],
        [ new TEntry('field_19') ],
        [ new TEntry('field_20') ],
        [ new TEntry('field_21') ] );
    $row->layout = ['col-sm-2 control-label', 'col-sm-2', 'col-sm-6', 'col-sm-2'];

    // ...
    $this->form->addAction('Send', new TAction(array($this, 'onSend')),
        'fa:check-circle-o green');

    // empacota o conteúdo em uma caixa vertical
    $vbox = new TVBox;
    $vbox->style = 'width: 100%';
    $vbox->add(new TXMLBreadCrumb('menu.xml', __CLASS__));
    $vbox->add($this->form);

    parent::__add($vbox);
}

}

```

Na próxima figura, podemos ver a distribuição automática de campos na tela.

Formulário Bootstrap colunas
MARCOS ANTONIO RAFAEL DA FONSECA -

Automatic form grid Manual form grid

2 slots

3 slots

4 slots

5 slots

6 slots

Send

MARCOS ANTONIO RAFAEL DA FONSECA -

Figura 42 Formulário Bootstrap em colunas – distribuição automática

Na próxima figura, podemos ver a distribuição manual de campos na tela.

Formulário Bootstrap colunas
MARCOS ANTONIO RAFAEL DA FONSECA -

Automatic form grid Manual form grid

Custom 1

Custom 2

Custom 3

Custom 4

Custom 5

Send

MARCOS ANTONIO RAFAEL DA FONSECA -

Figura 43 Formulário Bootstrap em colunas – distribuição manual

4.5.5 Formulário Bootstrap com labels acima

Nos formulários desenvolvidos anteriormente, você deve ter percebido que os rótulos dos campos normalmente ficam alinhados horizontalmente e geralmente à esquerda dos campos. Entretanto, em algumas situações, é desejável ter os rótulos acima dos campos, para melhor aproveitamento de espaço ou por escolha de design do projeto.

Neste exemplo, vamos construir um formulário Bootstrap responsivo, em que os rótulos de texto ficam acima dos campos. Para tal, também vamos utilizar a classe `BootstrapFormBuilder` para a montagem do formulário. No método construtor, criamos o formulário, definimos seu título, e utilizamos o método `setFieldSizes()`. Este método é muito importante para que nossa abordagem funcione. Da maneira que está

sendo utilizado, ele define o tamanho de todos os campos do formulário, inclusive rótulos de texto. Assim, todos elementos ocuparão 100% do espaço disponível no slot. Como os rótulos (labels) dividirão espaço dos slots com os campos, automaticamente eles empurrarão os campos para baixo, dividindo espaço na vertical.

Após estas definições, adicionamos uma aba ao formulário, pelo método `appendPage()`, e instanciamos alguns objetos de entrada de dados. Logo em seguida, definimos propriedades como máscaras de preenchimento de valores, adicionamos opções aos campos do tipo combo, dentre outros.

Ao utilizarmos o método `addFields()`, perceba que estamos utilizando o conceito de slots, mas dentro de cada slot existem geralmente dois elementos: um rótulo, e um campo de entrada de dados. Após utilizarmos o `addFields()`, definimos as proporções dos slots ao atribuirmos as classes `col-sm-x` na propriedade `layout`. Repetimos isto para cada linha a ser adicionada ao formulário. Assim, cada linha conterá pares de rótulo de texto, e campo, que ficarão um abaixo do outro por ocuparem 100% da largura disponível, o que é garantido pelo método `setFieldSizes()`.

app/control/Presentation/Forms/FormVerticalBuilderView.class.php

```
<?php
class FormVerticalBuilderView extends TPage
{
    private $form;

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

        // cria o formulário
        $this->form = new BootstrapFormBuilder('form_builder');
        $this->form->setFormTitle(_t('Bootstrap vertical form'));
        $this->form->generateAria(); // automatic aria-label

        // define o tamanho par aos campos do formulário
        $this->form->setFieldSizes('100%');

        // adiciona aba ao formulário
        $this->form->appendPage('Basic');

        // criação dos objetos
        $code      = new TEntry('code');
        $name     = new TEntry('name');
        $doc      = new TEntry('doc');
        $gender   = new TCombo('gender');
        $driver   = new TEntry('driver');
        $birthdate = new TDate('birthdate');
        $status   = new TCombo('status');
        $homephone = new TEntry('homephone');
        $cellphone = new TEntry('cellphone');
        $street   = new TEntry('street');
        $number   = new TEntry('number');
        $neighborhood = new TEntry('neighborhood');
        $city     = new TDBUniqueSearch('city', 'samples', 'City', 'id', 'name');
        $state   = new TCombo('state');
        $postal  = new TEntry('postal');
```

```

// definição de propriedades dos campos
$cidade->setMinLength(1);
$cidade->setMask('{{id}} <b>{name}</b> - {state->name}');
$status->addItems( ['S' => 'Single', 'C' => 'Committed' ] );
$gênero->addItems( ['F' => 'Female', 'M' => 'Male' ] );
$estado->addItems([ 'AL' => 'Alabama', 'AK' => 'Alaska', 'AZ' => 'Arizona',
                    'CA' => 'California', 'CO' => 'Colorado' ]);
$postal->setMask('99.999.999', true);
$documento->setMask('999.999.999-99');
$dataNascimento->setMask('dd/mm/yyyy');
$telefoneResidencial->setMask('(99)9999-9999');
$celular->setMask('(99)9999-9999');
$dataNascimento->setDatabaseMask('yyyy-mm-dd');
$estado->enableSearch();

// adiciona uma linha com rótulos e campos
$row = $this->form->addField( [ new TLabel('Code'),      $code ],
                                [ new TLabel('Name'),       $name ],
                                [ new TLabel('Gender'),    $gender ],
                                [ new TLabel('Status'),   $status ] );
$row->layout = ['col-sm-2', 'col-sm-6', 'col-sm-2', 'col-sm-2'];

// adiciona uma linha com rótulos e campos
$row = $this->form->addField( [ new TLabel('Driver lic.'), $driver ],
                                [ new TLabel('Document'), $documento ],
                                [ new TLabel('Birthdate'), $dataNascimento ],
                                [ new TLabel('Home phone'), $telefoneResidencial ],
                                [ new TLabel('Cell phone'), $celular ] );
$row->layout = ['col-sm-2', 'col-sm-3', 'col-sm-3', 'col-sm-2', 'col-sm-2'];

// adiciona um label de divisão no formulário
$label2 = new TLabel('Address', '#5A73DB', 12, '');
$label2->style='text-align:left; border-bottom:1px solid #c0c0c0; width:100%';
$this->form->addContent( [$label2] );

// adiciona uma linha com rótulos e campos
$row = $this->form->addField( [ new TLabel('Street.'),      $rua ],
                                [ new TLabel('Number'),     $numero ],
                                [ new TLabel('Neighborhood'), $bairro ] );
$row->layout = ['col-sm-6', 'col-sm-2', 'col-sm-4'];

$row = $this->form->addField( [ new TLabel('City.'),      $cidade ],
                                [ new TLabel('State'),     $estado ],
                                [ new TLabel('Postal'),   $postal ] );
$row->layout = ['col-sm-6', 'col-sm-3', 'col-sm-3'];

// adiciona outra aba ao formulário
// ...

$this->form->addAction('Send', new TAction(array($this, 'onSend')),
                        'fa:check-circle-o green');

// empacota tudo usando uma caixa vertical
$vbox = new TVBox;
$vbox->style = 'width: 100%';
$vbox->add(new TXMLBreadCrumb('menu.xml', __CLASS__));
$vbox->add($this->form);

parent::add($vbox);
}

```

O formulário possui uma ação, criada pelo método `addAction()`, que por sua vez está conectada ao método `onSend()`. Este método somente apresenta os dados do formulário em tela para conferência.

```

public function onSend($param)
{
    $data = $this->form->getData();
    $this->form->setData($data);

    echo '<pre>';
    print_r($data);
    echo '</pre>';
}

```

A figura a seguir demonstra o formulário criado em execução.

The screenshot shows a web form titled "Formulário Bootstrap vertical". At the top right, it says "MARCOS ANTONIO RAFAEL DA FONSECA -". Below the title, there are two tabs: "Basic" (selected) and "Other data". The "Basic" tab contains fields for "Code" (input), "Name" (input), "Gender" (dropdown), and "Status" (dropdown). The "Other data" tab contains fields for "Driver lic." (input), "Document" (input), "Birthdate" (input with a calendar icon), "Home phone" (input), and "Cell phone" (input). Below these tabs is a section titled "Address" with fields for "Street" (input), "Number" (input), and "Neighborhood" (input). Under "Address" is a section for "City" with a dropdown menu showing "Buscar" and a "State" dropdown menu showing "Selecionar". To the right of these is a "Postal" input field. At the bottom left is a "Send" button with a green checkmark icon. At the bottom right, it says "MARCOS ANTONIO RAFAEL DA FONSECA -".

Figura 44 Formulário Bootstrap com rótulos acima

4.5.6 Postagem estática de formulários

Quando enviamos os dados de um formulário por meio de um botão (Salvar, Enviar), somente o núcleo da página é recarregado, e não toda a página. Com isso, a navegação fica bastante fluida. Como o núcleo da página é recarregado, isso nos permite adicionar novos componentes à tela após a postagem ou realizar possíveis atualizações de valores em campos após a postagem da ação. Como por exemplo, podemos citar o campo ID, que geralmente é preenchido após a gravação de um registro novo.

Uma outra abordagem possível no Framework é a de não recarregar o núcleo da página após a postagem dos dados, fazendo com que somente o método específico de salvamento seja executado, o que chamamos de postagem estática, por executar somente um método estático, e não passar novamente pelo construtor. A postagem estática enviará os dados para o método, mas não recarregará o núcleo da página. O lado positivo é que serão transmitidos menos dados na requisição. O lado negativo é que não podemos usar a variável `$this` dentro de um método estático, para alterar um objeto criado no construtor, por exemplo. Assim, devemos coletar os dados por meio do parâmetro `$param`, que é do tipo Array.

No exemplo a seguir, pegamos o exemplo `FormStaticBuilderView`, que vem junto com o tutor, e marcamos o método `onSave()` como `STATIC`. Isso fará com que somente os dados do formulário sejam postados para o método `onSave()`. Com isso, nenhuma parte da tela será recarregada, nem ao menos o formulário, como tradicionalmente ocorre. Neste caso, não podemos usar a variável `$this` para chamar o `$this->form->getData()`, mas podemos obter todos os dados a partir da variável `$param`, que é um vetor. Dentro de um método estático podemos criar diálogos (`TMessage`) e também podemos invocar métodos estáticos como `TForm::sendData()`, que serve para enviar dados ao formulário por meio de JavaScript.

`app/control/Presentation/Forms/FormStaticBuilderView.class.php`

```
<?php
class FormStaticBuilderView extends TPage
{
    private $form;

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

        $this->form = new BootstrapFormBuilder;
        $this->form->setFormTitle(_t('Form with static post'));

        $id      = new TEntry('id');
        $name   = new TEntry('name');
        $address = new TEntry('address');
        $date   = new TDate('date');
        $obs     = new TText('obs');

        // adiciona algumas linhas ao formulário
        $this->form->addFields( [ new TLabel('Id') ], [ $id ] );
        $this->form->addFields( [ $lbl=new TLabel('Name') ], [ $name ] );
        $this->form->addFields( [ new TLabel('Address') ], [ $address ] );
        $this->form->addFields( [ new TLabel('Date') ], [ $date ] );
        $this->form->addFields( [ new TLabel('Obs') ], [ $obs ] );

        $lbl->setFontColor('red');
        $id->setSize('30%');
        $name->setSize('70%');
        $address->setSize('70%');
        $date->setSize('30%');
        $obs->setSize('70%');

        $this->form->addAction('Send', new TAction(array($this, 'onSend')),
                               'fa:check-circle-o green');

        parent::add($this->form);
    }
}
```

O método `onSend()` é executado pelo botão de ação deste formulário, o que foi determinado pela chamada de `addAction()`. Perceba que ele é marcado como `static`. Sempre que uma ação estiver vinculada a um método estático, somente este método será executado, não passando novamente pelo construtor. Assim, a página não é recarregada. Apesar disso, ações como exibição de diálogo de mensagens (`TMessage`), ou envio de dados para o formulário (`TForm::sendData()`) podem ser utilizados.

```

public static function onSend($param)
{
    try
    {
        // valida campo vazio
        if (empty($param['name']))
        {
            throw new Exception('Name cannot be empty');
        }

        // apresenta resultados
        new TMessage('info', str_replace("\n", '<br> ', print_r($param, true)));
    }
    catch (Exception $e)
    {
        new TMessage('error', $e->getMessage());
    }
}
}

```

A figura a seguir apresenta o resultado da execução deste programa.

The screenshot shows a web-based form titled "Formulário com post estático". At the top, it displays the name "MARCOS ANTONIO RAFAEL DA FONSECA". Below the title, there are five input fields: "Id" (containing "1"), "Name" (containing "2"), "Address" (containing "3"), "Date" (containing "2018-07-04" with a calendar icon), and "Obs" (containing "5"). At the bottom left is a "Send" button with a circular arrow icon, and at the bottom right is the same name "MARCOS ANTONIO RAFAEL DA FONSECA".

Figura 45 Formulário com post estático

Use a postagem estática de dados em formulários, sempre que você não quiser que absolutamente nada relacionado ao estado da tela se altere, por que quando a requisição é completada, o formulário não é reconstruído. Use também quando quiser que o tamanho da requisição HTTP de retorno seja o menor possível, tendo em vista que neste cenário, somente retornará pela requisição, mensagens de sucesso, erro, exceções, e não todo o formulário (HTML) novamente para ser renderizado.

4.5.7 Formulário com lista de campos

Na maioria das vezes em que construímos um formulário, sabemos exatamente quantos e quais serão os seus campos. Os formulários construídos até o momento foram assim, sendo que quando criamos os campos, definimos nomes específicos (Ex: `id`, `nome`, `descricao`, `endereco`), que foram utilizados também para ler seus valores em métodos de coleta de dados (`onSend`), acionados a partir do botão de envio.

Entretanto, existem situações dinâmicas nas quais o usuário pode entrar com “N” ocorrências de uma determinada informação. Nestes casos, não podemos prever o número máximo de ocorrências. E mesmo que pudéssemos, teríamos que encher a tela com uma quantidade razoável de campos, para que o usuário prenchesse somente alguns deles. Não é interessante criarmos uma grande quantidade de campos para prever repetições (Ex: endereco1, endereco2, endereco3, rua1, rua2, rua3, cidade1, cidade2, cidade3), por que criariamos campos desnecessários, e mesmo criando muitos deles, em alguns casos ainda faltariam campos para o usuário preencher.

Como exemplo, podemos citar que um cliente pode ter “N” informações de endereços vinculados (endereço de entrega, de cobrança, endereço profissional), cada qual com vários campos (rua, número, CEP, cidade), ou “N” informações de dados de contato (nome, telefone, facebook, twitter, whatsapp, etc). Para estes casos, podemos criar um formulário que permita repetir um conjunto de campos conforme a necessidade, tratando-os como vetores. Assim, o usuário poderá excluir uma linha de campos, ou adicionar uma nova, conforme a quantidade de repetições que precisar.

O próximo exemplo demonstra a possibilidade de tratar campos como vetores para a postagem de várias ocorrências de um valor. Para tal, utilizamos a classe **TFieldList**, que é responsável por gerenciar listas com repetições de campos. Esta classe será utilizada dentro de um formulário **BootstrapFormBuilder**.

Nosso exemplo inicia com a criação do formulário (**BootstrapFormBuilder**). Logo em seguida, criamos os objetos de entrada de dados, como: **TCombo**, **TEntry**, e **TDate**. Note que na declaração do objeto, utilizamos a notação vetorial [], para indicar que aquele objeto pode ter mais ocorrências. Isto será necessário, pois a linha que contém o objeto poderá ser clonada várias vezes pelo usuário para entrada de valores repetidos.

Em seguida, é criado o objeto **TFieldList**, que é um container para objetos vetoriais que trata de seu gerenciamento, que envolve replicação e exclusão de linhas. O método **addField()** adiciona um campo vetorial à lista de campos. Já o método **enableSorting()** habilita a reordenação de linhas, que pode ser feita com o mouse usando clique e arraste. Ainda é necessário chamar o método **addField()** da classe **BootstrapFormBuilder**, para que esta classe saiba quais campos gerenciará na postagem.

O método **addHeader()** é responsável por criar a linha de cabeçalho, com as informações de títulos de colunas informadas pelo método **addField()**. Já o método **addDetail()** adiciona uma linha vazia com os campos vetoriais. Em princípio só é necessário chamar o método **addDetail()** uma vez, a não quer que desejamos deixar várias linhas disponíveis previamente para preenchimento. O botão **addCloneAction()** adiciona uma linha com ação para clonagem.

A lista de campos (`TFieldList`) precisa ser adicionada ao formulário para ser exibida. Ao final, é criado um botão de ação, e este é utilizado posteriormente para exibição dos resultados digitados pelo usuário.

app/control/Presentation/Forms/FormFieldListView.class.php

```
<?php
class FormFieldListView extends TPage
{
    private $form;
    private $fieldlist;

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

        // cria o formulário
        $this->form = new BootstrapFormBuilder('my_form');
        $this->form->setFormTitle(_t('Form field list'));

        $combo = new TCombo('combo[]');
        $combo->enableSearch();
        $combo->addItems(['1'=>'One', '2'=>'Two', '3'=>'Three', '4'=>'Four', '5'=>'Five']);
        $combo->setSize('100%');

        $text = new TEntry('text[]');
        $text->setSize('100%');

        $number = new TEntry('number[]');
        $number->setNumericMask(2,',','.', true);
        $number->setSize('100%');
        $number->style = 'text-align: right';

        $date = new TDate('date[]');
        $date->setSize('100%');

        $this->fieldlist = new TFieldList;
        $this->fieldlist->generateAria();
        $this->fieldlist->width = '100%';
        $this->fieldlist->name = 'my_field_list';
        $this->fieldlist->addField( '<b>Combo</b>', $combo, ['width' => '25%'] );
        $this->fieldlist->addField( '<b>Text</b>', $text, ['width' => '25%'] );
        $this->fieldlist->addField( '<b>Number</b>', $number, ['width' => '25%'] );
        $this->fieldlist->addField( '<b>Date</b>', $date, ['width' => '25%'] );

        $this->fieldlist->enableSorting();

        $this->form->addField($combo);
        $this->form->addField($text);
        $this->form->addField($number);
        $this->form->addField($date);

        $this->fieldlist->addHeader();
        $this->fieldlist->addDetail( new stdClass );
        $this->fieldlist->addCloneAction();

        // adiciona a field list ao form
        $this->form->addContent( [$this->fieldlist] );

        // adiciona ações
        $this->form->addAction( 'Save', new TAction([{$this}, 'onSave']), 'fa:save blue' );
    }
}
```

```
// empacota formulário em uma caixa vertical
$cbox = new TBox;
$cbox->style = 'width: 100%';
$cbox->add(new TXMLBreadCrumb('menu.xml', __CLASS__));
$cbox->add($this->form);

parent::add($cbox);
}
```

O método `onSave()` será utilizado somente para exibir em uma nova janela, todos os dados recebidos. É importante notar que este método é estático, o que faz com que a tela não seja recarregada e os campos mantém o seu valor mesmo após a postagem. A função `print_r()` é usada para converter o vetor de dados (`$param`), em string, desde que informado o booleano `TRUE`, como segundo parâmetro da função.

```
public static function onSave($param)
{
    // exibe valores dentro de uma janela
    $win = TWindow::create('test', 0.6, 0.8);
    $win->add( '<pre>' . str_replace("\n", '<br>', print_r($param, true)) . '</pre>' );
    $win->show();
}
```

Na próxima figura, podemos ver o resultado da execução deste programa.

Combo	Text	Number	Date
One	Part. One	10,10	2018-07-31
Two	Part. Two	20,20	2018-08-01
Three	Part. Three	30,30	2018-08-02
Four	Part. Four	40,40	2018-08-03
Five	Part. Five	50,50	2018-08-04

Figura 46 Formulário com múltiplos valores

4.5.8 Formulário com check list

Sempre que precisamos solicitar ao usuário selecionar uma ou várias opções, podemos usar o componente `TCheckGroup`, que oferece uma lista de caixas de seleção. Porém este componente limita o tamanho da informação a ser exibida. Em alguns casos, precisamos exibir não apenas um nome, mas várias colunas para que o usuário decida o que selecionar. Foi pensando nestas situações que foi criado o componente `TCheckList`.

O componente `TCheckList` permite criarmos uma lista de checagem com várias colunas, permitindo exibir mais detalhes para o usuário escolher o registro a selecionar. O formulário de exemplo a seguir não possui somente um componente `TCheckList`, mas também outros dois componentes de input: `TDBUniqueSearch` e `TDate`.

Para criar uma checklist, basta instanciar o componente `TCheckList` e indicar um nome em seu construtor. É por meio deste nome, que obteremos posteriormente os valores na postagem. O componente possui o método `addColumn()`, que adiciona colunas à checklist. Cada coluna tem um nome, rótulo, alinhamento e tamanho.

Valores podem ser adicionados à checklist da mesma maneira que adicionamos valores em umadatagrid, simplesmente adicionando objetos. Quando utilizamos o método `addColumn()` indicamos o nome de um atributo como primeiro parâmetro. Esta informação é utilizada para extrair de cada um dos objetos adicionados a informação a ser exibida na coluna. Neste exemplo, para simplificar, adicionamos todos os produtos, por meio do método `Product::allInTransaction('samples')`, que é um atalho do método `all()` que abre uma transação automática com a base `samples`.

`app/control/Presentation/Forms/FormCheckListView.class.php`

```
<?php
class FormCheckListView extends TPage
{
    private $form;

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

        $this->form = new BootstrapFormBuilder;
        $this->form->setFormTitle('Checklist');

        $customer = new TDBUniqueSearch('customer_id', 'samples', 'Customer', 'id',
'name');
        $date     = new TDate('order_date');
        $orderlist = new TCheckList('order_list');

        $customer->setMinLength(1);
        $orderlist->addColumn('id',           'Id',           'center',  '10%');
        $orderlist->addColumn('description', 'Description', 'left',    '50%');
        $orderlist->addColumn('sale_price',   'Price',       'left',    '40%');
        $orderlist->setHeight(250);
        $orderlist->makeScrollable();

        $this->form->addFields( [ new TLabel('Customer') ], [ $customer ] );
        $this->form->addFields( [ new TLabel('Date') ],      [ $date ] );

        $input_search = new TEntry('search');
        $input_search->placeholder = _t('Search');
        $input_search->setSize('100%');
        $orderlist->enableSearch($input_search, 'id, description');

        $hbox = new THBox;
        $hbox->style = 'border-bottom: 1px solid gray;padding-bottom:10px';
        $hbox->add( new TLabel('Order list') );
        $hbox->add( $input_search )->style = 'float:right;width:30%;';

        // load order items
        $orderlist->addItems( Product::allInTransaction('samples') );
        $this->form->addContent( [$hbox] );
        $this->form->addFields( [$orderlist] );

        $this->form->addAction( 'Save', new TAction([ $this, 'onSave' ]), 'fa:save
green' );
    }
}
```

```
// empacota conteúdo usando caixa vertical.
$cbox = new TBox;
$cbox->style = 'width: 100%';
$cbox->add(new TXMLBreadCrumb('menu.xml', __CLASS__));
$cbox->add($this->form);

parent::add($cbox);
}
```

Quando o usuário clicar no botão de salvar, o método `onSave()` será executado, exibindo o conteúdo preenchido no formulário, bem como as opções marcadas na checklist.

```
public function onSave($param)
{
    $data = $this->form->getData();

    echo '<pre>';
    var_dump($data);
    echo '</pre>';

    // mantém form preenchido
    $this->form->setData($data);
}
```

A figura a seguir demonstra um checklist.

ID	Description	Price
1	Pendrive 512Mb	57.6
2	HD 120 GB	180.0
3	SD CARD 512MB	35.0
4	SD CARD 1GB MINI	40.0
5	CAM. PHOTO I70 Silver	900.0
6	CAM. PHOTO DSC-W50 Silver	700.0

Figura 47 Formulário com checklist

4.5.9 Estilos de botão

Como o Framework possui algumas bibliotecas como a Bootstrap e Font Awesome, que possuem coleções de ícones, de maneira integrada, é possível utilizar os seus recursos para botões de formulários e listagens. Neste exemplo, será demonstrado como criar ícones utilizando recursos dessas bibliotecas.

No exemplo a seguir, inicialmente estamos criando um botão (`$button_gi`), e utilizando o método `setImage()` para definir seu ícone. Neste momento, podemos utilizar o prefixo “`bs:`” para indicar uma classe CSS da biblioteca Bootstrap, que fornecerá o ícone correspondente, que nesse caso foi “`remove-circle`” com a cor azul. No segundo exemplo, criamos um botão (`$button_fa`), e utilizamos o método `setImage()` para definir o ícone por meio da biblioteca Font Awesome. Neste caso, basta utilizarmos o prefixo “`fa:`” seguido da classe CSS, que neste caso foi “`trash-o`”, seguida da cor utilizada. Mais informações sobre os ícones disponíveis podem ser obtidas nos sites de cada biblioteca.

Em seguida, é demonstrado como criar um botão (`$button_bs`) utilizando uma classe nativa da Bootstrap. Neste caso, basta atribuirmos a classe CSS do botão (atributo `class`). Neste exemplo, foi utilizada a classe “`btn-success`”.

No último exemplo, criamos um botão (`$button_po`), para demonstrar a utilização do Popover. O Popover é um recurso da biblioteca Bootstrap que permite criar uma camada de informação ao passarmos o mouse sobre um elemento. Para habilitar o Popover sobre um objeto do framework, basta definirmos ‘`true`’ para o atributo `popover`. O título e o conteúdo do Popover são definidos pelos atributos `ptitle`, e `popcontent`. Ainda podemos usar o atributo `poside` para definir o lado no qual o Popover abrirá (`top`, `bottom`, `right`, `left`).



Figura 48 Botões com estilos diversos

app/control/Presentation/Forms/FormButtonStyleView.class.php

```
<?php
class FormButtonStyleView extends TPage
{
    public function __construct()
    {
        parent::__construct();
        $hbox = new THBox; // caixa horizontal
```

```

// botão com Font Awesome
$button_fa = new TButton('button_fa');
$button_fa->setLabel('FA trash');
$button_fa->setImage('fa:trash-o red');
$hbox->add( $button_fa );

// botão com classe Bootstrap
$button_bs = new TButton('button_bs');
$button_bs->setLabel('Success');
$button_bs->class = 'btn btn-success';
$hbox->add( $button_bs );

// botão com popover
$button_po = new TButton('button_po');
$button_po->setLabel('Popover');
$button_po->popover = 'true';
$button_po->poptitle = 'Pop title';
$button_po->popcontent = 'This is the <br>popover for button 1';
$hbox->add( $button_po );

parent::add($hbox);
}
}

```

Obs: Na aplicação Tutor, você encontrará um exemplo mais completo deste recurso.

4.5.10 Máscaras de input

Neste exemplo, abordaremos a utilização de diferentes tipos de máscaras de digitação para entrada de dados. A classe `TEntry` possui o método `setMask()`, que permite definir uma máscara de digitação. Para definir uma máscara de digitação de CEP por exemplo, basta executarmos `setMask('99.999-999')`. Porém, o método `setMask()` também permite diferentes combinações para filtrar a digitação de letras e números. O exemplo a seguir nos mostra diferentes combinações aceitas. Ao utilizarmos “A!” por exemplo, estamos indicando a digitação somente de letras e números. Já “AAA” permite apenas 3 caracteres alfanuméricos. A máscara “S!” permite apenas letras. A máscara “SSS” permite apenas 3 letras. A máscara “9!” permite apenas números. A máscara “999” permite apenas 3 números, e a máscara “SSS-9999” permite 3 letras e 4 números. Por fim, o método `forceUpperCase()` garante a digitação em maiúsculo, e o método `forceLowerCase()` garante a digitação em minúsculo.

O método `setMask()` também possui um segundo parâmetro (`boolean`), que ao ser indicado `true`, remove a máscara após o envio dos dados. Assim, você obtém o dado sem os caracteres de pontuação. Quando este parâmetro for `true`, o `setValue()`, usado para preencher o objeto com valores, não precisará passar o dado com a pontuação. Note que o `$element1` e o `$element3`, postam e recebem valores com a máscara. Já o `$element2`, e `$element4`, postam e recebem valores sem a máscara.

Obs: Alguns trechos foram cortados na edição em virtude do espaço, mas encontram-se integralmente junto ao código-fonte do Tutor.

app/control/Presentation/Forms/FormMaskView.class.php

```
<?php
class FormMaskView extends TPage
{
    private $form;

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

        // cria o formulário
        $this->form = new BootstrapFormBuilder;
        $this->form->setFormTitle(_t('Input masks'));

        // cria vários campos de input
        $element1 = new TEntry('element1');
        $element2 = new TEntry('element2');
        $element3 = new TEntry('element3');
        $element4 = new TEntry('element4');
        $element5 = new TEntry('element5');
        $element6 = new TEntry('element6');
        $element7 = new TEntry('element7');
        $element8 = new TEntry('element8');
        $element9 = new TEntry('element9');
        $element10= new TEntry('element10');
        $element11= new TEntry('element11');
        $element12= new TEntry('element12');
        $element13= new TEntry('element13');
        $element14= new TEntry('element14');

        // configura as máscaras
        $element1->setMask('99.999-999');
        $element2->setMask('99.999-999', true);
        $element3->setMask('99.999.999/9999-99');
        $element4->setMask('99.999.999/9999-99', true);
        $element5->setMask('A!');
        $element6->setMask('AA');
        $element7->setMask('S!');
        $element8->setMask('SSS');
        $element9->setMask('9!');
        $element10->setMask('999');
        $element11->setMask('SSS-9999');
        $element12->forceUpperCase();
        $element13->forceLowerCase();
        $element14->setNumericMask(2, ',', '.', true);

        // define valores iniciais para os campos
        $element1->setValue('95.900-716');
        $element2->setValue('95900716');
        $element3->setValue('05.343.117/0001-44');
        $element4->setValue('05343117000144');

        // ajusta as proporções das colunas
        $this->form->setColumnClasses(2, ['col-sm-4', 'col-sm-8']);

        // adiciona os campos ao formulário
        $this->form->addFields( [new TLabel('Element 1 (99.999-999)'), [$element1] ]);
        // ...
        parent::__add($this->form);
    }
}
```

4.5.11 Validações

Os exemplos anteriores nos permitiram dominar as técnicas básicas de montagem de formulários. Também vimos como criar ações e obter os dados do formulário. Não basta somente obtermos os dados do formulário, é preciso que os mesmos sejam válidos. Para garantir que os dados obtidos de um formulário sejam válidos, o Framework possui uma série de validadores para campos de formulários.

Para acrescentar validação a um formulário, basta utilizarmos o método `addValidation()` sobre o campo que desejamos validar e, no momento de obter os dados do formulário, executar o método `validate()`. Inicialmente, para cada campo do formulário que deve passar por algum tipo de validação, deve ser executado o método `addValidation()`. Este método recebe como parâmetros: um nome para descrever o campo (a ser utilizado em mensagens de validação); um objeto de validação (Ex: `TMinValidator`); e parâmetros específicos para aquele tipo de validação. Veja na tabela a seguir os validadores que acompanham o Framework.

Tabela 3. Validadores

Validador	Descrição
<code>TRequiredValidator</code>	Valida se o campo foi preenchido (obrigatoriedade).
<code>TNumericValidator</code>	Valida se o campo contém um número válido.
<code>TMinLengthValidator</code>	Valida a quantidade mínima de caracteres do campo.
<code>TMaxLengthValidator</code>	Valida a quantidade máxima de caracteres do campo.
<code>TMinValueValidator</code>	Valida o valor mínimo que deve estar presente no campo.
<code>TMaxValueValidator</code>	Valida o valor máximo que deve estar presente no campo.
<code>TEmailValidator</code>	Valida se o campo contém um e-mail válido.
<code>TCNPJValidator</code>	Valida se o campo contém um CNPJ válido.
<code>TCPFValidator</code>	Valida se o campo contém um CPF válido.

O objetivo desse exemplo, que pode ser visto no código-fonte a seguir, é criar um formulário que demonstre alguns dos principais tipos de validações que podemos acrescentar a um formulário. Assim, criamos um formulário (`BootstrapFormBuilder`) contendo em seu interior uma série de campos com validações distintas.

Após criar o formulário, criamos vários campos (`$field1`, `$field2`, etc) sobre os quais serão adicionadas validações. Em seguida, acionamos os campos ao formulário com o método `addField()`. Posteriormente, utilizamos o método `addValidation()`, para definir o tipo de validação que cada campo terá (conforme tabela 3). Alguns validadores recebem parâmetros adicionais, como: `TMinLengthValidator`, para definir qual o tamanho mínimo do campo; ou `TMinValueValidator`, para definir qual o valor mínimo do campo.

Após definir as validações, criamos a ação do formulário, pelo método `addAction()`, que criará um botão de ação vinculado ao método `onSend()`.

app/control/Presentation/Forms/FormValidationView.class.php

```
<?php
class FormValidationView extends TPage
{
    private $form;

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

        // cria o formulário
        $this->form = new BootstrapFormBuilder;
        $this->form->setFormTitle(_t('Form validation'));
        $this->form->setClientValidation(true); // liga a validação client side

        // cria os campos do formulário
        $field1 = new TEntry('field1');
        $field2 = new TEntry('field2');
        $field3 = new TEntry('field3');
        $field4 = new TEntry('field4');
        $field5 = new TEntry('field5');
        $field6 = new TEntry('field6');
        $field7 = new TEntry('field7');

        // adiciona os campos ao formulário
        $this->form->addFields([new TLabel('1. Min length validator (3)'), [$field1]];
        $this->form->addFields([new TLabel('2. Max length validator (20)'), [$field2]];
        $this->form->addFields([new TLabel('3. Min value validator (1)'), [$field3]];
        $this->form->addFields([new TLabel('4. Max value validator (10)'), [$field4];
        $this->form->addFields([new TLabel('5. Required validator:'), [$field5];
        $this->form->addFields([new TLabel('6. Email validator:'), [$field6];
        $this->form->addFields([new TLabel('7. Numeric validator:'), [$field7];

        // adiciona as validações
        $field1->addValidation('Field 1', new TMinLengthValidator, array(3));
        $field2->addValidation('Field 2', new TMaxLengthValidator, array(20));
        $field3->addValidation('Field 3', new TMinValueValidator, array(1));
        $field4->addValidation('Field 4', new TMaxValueValidator, array(10));
        $field5->addValidation('Field 5', new TRequiredValidator);
        $field6->addValidation('Field 6', new TEmailValidator);
        $field6->addValidation('Field 6', new TRequiredValidator);
        $field7->addValidation('Field 7', new TNumericValidator);

        // adiciona a ação ao formulário
        $this->form->addAction('Send', new TAction(array($this, 'onSend')),
            'fa:save green');
        parent::add($this->form);
    }
}
```

Obs: Veja que no caso do `$field6`, adicionamos duas validações.

Quando o botão for clicado, o método `onSend()` é disparado. Neste momento, obtemos os dados do formulário por meio do método `getData()`. Além disso, é executado o método `validate()`, que rodará cada um dos validadores acrescentados pelo método `addValidation()`. Todo o bloco está contido dentro de um `try/catch` pois se alguma validação não for executada com êxito, uma exceção será lançada. Neste caso, a validação é tratada no bloco `catch` com uma mensagem de erro.

```

public function onSend($param)
{
    try
    {
        // obtém os dados do formulário
        $data = $this->form->getData();

        // roda as validações
        $this->form->validate();

        // cria uma string com os valores dos campos
        $message = 'Field 1 : ' . $data->field1 . '<br>';
        $message.= 'Field 2 : ' . $data->field2 . '<br>';
        $message.= 'Field 3 : ' . $data->field3 . '<br>';
        $message.= 'Field 4 : ' . $data->field4 . '<br>';
        $message.= 'Field 5 : ' . $data->field5 . '<br>';
        $message.= 'Field 6 : ' . $data->field6 . '<br>';
        $message.= 'Field 7 : ' . $data->field7 . '<br>';

        // exibe a mensagem
        new TMessage('info', $message);
    }
    catch (Exception $e)
    {
        // exibe a mensagem de exceção, caso alguma validação falhe
        new TMessage('error', $e->getMessage());
    }
}

```

Na próxima figura, podemos ver o resultado da execução deste programa.

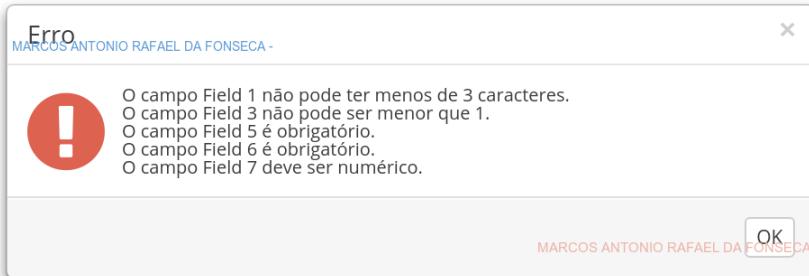


Figura 49 Validação de formulário

4.5.12 Criando um validador

Por mais validadores prontos que existam, jamais conseguiríamos atender todas as necessidades que surgem durante o desenvolvimento de um sistema. Nesse caso, é preciso saber como criar um validador novo, o que veremos neste exemplo. No exemplo demonstrado a seguir, criaremos um validador de datas (**TDateValidator**). Um validador recebe por padrão como parâmetros: um rótulo de texto que identifica o nome do campo não validado na mensagem exceção; o valor postado do campo pelo formulário; e um vetor de parâmetros opcional.

app/lib/validator/TDateValidator.class.php

```
<?php
class TDateValidator extends TFieldValidator
{
    /**
     * Valida uma data
     * @param $label Identifica o campo, em caso de exceção
     * @param $value Valor a ser validado
     * @param $parameters Parâmetros adicionais (ex: máscara)
    */
    public function validate($label, $value, $parameters = NULL)
    {
        $mask      = $parameters[0];
        $year_pos  = strpos($mask, 'yyyy');
        $month_pos = strpos($mask, 'mm');
        $day_pos   = strpos($mask, 'dd');

        $year      = substr($value, $year_pos, 4);
        $month     = substr($value, $month_pos, 2);
        $day       = substr($value, $day_pos, 2);

        if (!checkdate((int) $month, (int) $day, (int) $year))
        {
            throw new Exception("O campo $label não é uma data válida ($mask)");
        }
    }
}
```

Obs: Um validador criado pelo usuário deve ser salvo em **app/lib/validator**. Confira outras regras de extensibilidade no site <http://www.adianti.com.br/framework-extensibility>.

O método **validate()** trata de realizar o processo de validação sobre o parâmetro **\$value**. Para validar uma data é preciso saber sua máscara de validação (Ex: **yyyy-mm-dd**). A máscara de validação é um parâmetro passado no momento em que instanciamos o objeto **TDateValidator**, e adicionamos a regra de validação ao objeto, o que será visto a seguir. Reconhecendo a máscara, então extraímos partes da data (dia, mês e ano), e usamos a função **checkdate()** do PHP para verificar se é uma data válida. Caso a data não seja válida, uma exceção deve ser lançada.

É importante lembrar que a exceção é lançada somente quando chamamos o método **validate()** do formulário, o que ocorre quando os dados são obtidos, geralmente junto ao método **getData()**.

A seguir, temos um exemplo de chamada para este validador sobre um campo qualquer (**\$field7**). Assim, usamos o método **addValidation()** para adicionar a validação, passando o rótulo do campo como primeiro parâmetro, o validador como segundo, e um vetor de parâmetros como terceiro.

```
<?php
...
$field7->addValidation('Data de Nascimento', new TDateValidator, array('dd/mm/yyyy'));
...
```

4.5.13 Seleções estáticas

Até agora vimos alguns componentes bastante utilizados em formulários como `TEntry`, `TText`, `TDate`, dentre outros. O objetivo deste e dos exemplos seguintes, é demonstrar especificamente os componentes de seleção de valores a partir de listas, como é o caso das seleções em forma de: botão de rádio (`TRadioGroup`); botão de checagem (`TCheckGroup`); combo box (`TCombo`); lista de seleção (`TSelect`); busca única (`TUniqueSearch`); busca múltipla (`TMultiSearch`); e, autocomplete (`TEntry`). Estes elementos possuem algumas características em comum, pois permitem seleção e busca de valores, e por isso serão agrupados neste mesmo exemplo.

Conforme pode ser visto no código-fonte a seguir, para exemplificar esses componentes, criaremos uma página contendo um formulário (`BootstrapFormBuilder`) e diversos tipos de campos de seleção (`TRadioGroup`, `TCheckGroup`, `TCombo`, `TSelect`, `TUniqueSearch`, `TMultiSearch`, e `Tentry`).

Após a criação dos componentes, utilizamos métodos como `setLayout()` em `TRadioGroup` e `TCheckGroup` para definir a orientação das opções (lado a lado, ou uma acima da outra); o método `setUseButton()`, para transformar as opções de radios e checks na forma de botões; o método `enableSearch()`, para transformar uma `TCombo` em um elemento que permite localização por digitação de valores; o método `setMaxSize()`, que define um limite para seleção de elementos em um `TMultiSearch`; e o método `setMinLength()`, que define a quantidade mínima de caracteres para iniciar a busca.

Após definirmos características dos elementos, usamos o `addItems()`, para adicionar uma lista de valores, indexada pela chave a ser enviada, e contendo em cada posição, o texto a ser exibido junto a cada opção. O método `setCompletion()` define uma lista de valores para autocomplete em um `TEntry`. Após, utilizamos o método `setValue()` para definir as opções previamente marcadas para cada componente. Por fim, um botão de ação é criado a fim de acionar o método `onSend()`.

Obs: O índice do vetor de opções do elemento de seleção corresponderá ao conteúdo enviado na postagem do formulário e o valor, ao conteúdo visualizado pelo usuário.

app/control/Presentation/Forms/FormStaticSelectionView.class.php

```
<?php
class FormStaticSelectionView extends TPage
{
    private $form;

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

        // cria o formulário
        $this->form = new BootstrapFormBuilder;
        $this->form->setFormTitle(_t('Static selections'));
```

```

// cria os campos
$radio = new TRadioGroup('radio');
$radio2 = new TRadioGroup('radio2');
$check = new TCheckGroup('check');
$check2 = new TCheckGroup('check2');
$combo = new TCombo('combo');
$combo2 = new TCombo('combo2');
$select = new TSelect('select');
$search = new TMultiSearch('search');
$unique = new TUniqueSearch('unique');
$multi = new TMultiEntry('multi');
$autocomp = new TEntry('autocomplete');

// define algumas propriedades
$radio->setLayout('horizontal');
$radio2->setLayout('horizontal');
$check->setLayout('horizontal');
$check2->setLayout('horizontal');
$radio2->setUseButton();
$check2->setUseButton();
$combo2->enableSearch();
$search->setMinLength(1);
$unique->setMinLength(1);
$search->setMaxSize(3);
$multi->setMaxSize(3);

// adiciona itens
$items = ['a'=>'Item a', 'b'=>'Item b', 'c'=>'Item c'];
$radio->addItems($items);
$check->addItems($items);
$radio2->addItems($items);
$check2->addItems($items);
$combo->addItems($items);
$combo2->addItems($items);
$select->addItems($items);
$search->addItems($items);
$unique->addItems($items);
$autocomp->setCompletion( array_values( $items ) );

// define valores default dos componentes
$radio->setValue('b');
$radio2->setValue('b');
$check->setValue( array('a', 'c') );
$check2->setValue( array('a', 'c') );
$combo->setValue('b');
$combo2->setValue('b');
$select->setValue(array('a', 'b'));
$search->setValue(array('a', 'c'));
$unique->setValue(array('b'));
$multi->setValue(array('aaa','bbb'));

// Ajusta as proporções dos slots das linhas do formulário
$this->form->setColumnClasses(2, ['col-sm-4', 'col-sm-8']);

// adiciona campos ao formulário
$this->form->addField( [new TLabel('TRadioGroup:'), [$radio] ] );
$this->form->addField( [new TLabel('TCheckGroup:'), [$check] ] );
$this->form->addField( [new TLabel('TRadioGroup (use button):'), [$radio2] ] );
$this->form->addField( [new TLabel('TCheckGroup (use button):'), [$check2] ] );
$this->form->addField( [new TLabel('TCombo:'), [$combo] ] );
$this->form->addField( [new TLabel('TCombo (with search):'), [$combo2] ] );
$this->form->addField( [new TLabel('TSelect:'), [$select] ] );
$this->form->addField( [new TLabel('TMultiSearch:'), [$search] ] );
$this->form->addField( [new TLabel('TUniqueSearch:'), [$unique] ] );
$this->form->addField( [new TLabel('TMultiEntry:'), [$multi] ] );
$this->form->addField( [new TLabel('Autocomplete:'), [$autocomp] ] );

```

```

// cria a ação de envio
$this->form->addAction('Send', new TAction(array($this, 'onSend')),
    'fa:check-circle-o green');

parent::add($this->form);
}

```

Quando o botão de ação do formulário é clicado, o método `onSend()` é executado. Neste momento, os dados do formulário são obtidos e uma string é montada para exibir ao usuário os valores selecionados por meio da classe `TMessage`.

```

public function onSend($param)
{
    $data = $this->form->getData(); // obtém os dados do formulário
    $this->form->setData($data); // mantém o form preenchido

    // cria uma string com os valores selecionados
    $message = 'Radio : ' . $data->radio . '<br>';
    $message.= 'Check : ' . print_r($data->check, TRUE) . '<br>';
    $message.= 'Radio (button) : ' . $data->radio2 . '<br>';
    $message.= 'Check (button) : ' . print_r($data->check2, TRUE) . '<br>';
    $message.= 'Combo : ' . $data->combo . '<br>';
    $message.= 'Combo2 : ' . $data->combo2 . '<br>';
    $message.= 'Select: ' . print_r($data->select, TRUE) . '<br>';
    $message.= 'Search: ' . print_r($data->search, TRUE) . '<br>';
    $message.= 'Unique: ' . print_r($data->unique, TRUE) . '<br>';
    $message.= 'Multi: ' . print_r($data->multi, TRUE) . '<br>';
    $message.= 'Autocomplete: ' . $data->autocomplete;

    new TMessage('info', $message); // exibe a mensagem
}
}

```

Na próxima figura, podemos ver o resultado da execução deste programa.

The screenshot displays a web-based form titled "Seletores estáticos" (Static Selectors). At the top, it says "MARCOS ANTONIO RAFAEL DA FONSECA -". Below the title, there are several input fields demonstrating different selector types:

- TRadioGroup:** A group of three radio buttons labeled "Item a", "Item b", and "Item c". "Item b" is checked.
- TCheckGroup:** A group of three checkboxes labeled "Item a", "Item b", and "Item c". "Item a" and "Item c" are checked.
- TRadioGroup (use button):** A group of three radio buttons labeled "Item a", "Item b", and "Item c". "Item b" is selected.
- TCheckGroup (use button):** A group of three checkboxes labeled "Item a", "Item b", and "Item c". "Item a" and "Item c" are selected.
- TCombo:** A dropdown menu containing the options "Item a", "Item b", and "Item c". "Item b" is selected.
- TCombo (with search):** A dropdown menu containing the options "Item a" and "Item b". "Item b" is selected.
- TSelect:** A dropdown menu containing the options "Item a", "Item b", and "Item c". "Item a" is selected.
- TMultiSearch:** A dropdown menu containing the options "Item a" and "Item c". Both are selected.
- TUniqueSearch:** A dropdown menu containing the option "Item b". It is selected.
- TMultiEntry:** A dropdown menu containing the options "aaa" and "bbb". Both are selected.
- Autocomplete:** An empty input field.

At the bottom left is a "Send" button, and at the bottom right is the text "MARCOS ANTONIO RAFAEL DA FONSECA -".

Figura 50 Formulário com seletores estáticos

4.5.14 Seleções manuais

No exemplo anterior, aprendemos a criar campos para seleção de elementos (**TRadioGroup**, **TCheckGroup**, **TCombo**, e **TSelect**) com valores definidos de maneira estática por meio de vetores. Entretanto, em muitas situações os valores que desejamos apresentar para o usuário selecionar estão no banco de dados. Neste caso, devemos coletar os valores diretamente na base de dados, antes de apresentá-los ao usuário.

O objetivo deste exemplo é apresentar os mesmos componentes de seleção vistos no exemplo anterior, mas agora alimentando-os diretamente com valores vindos de uma tabela na base de dados. A tabela que utilizaremos será a de categorias, que por sua vez é manipulada pelo Active Record **Category**. É o nome do Active Record e dos campos que precisamos para realizar esta operação.

Conforme pode ser acompanhado no código-fonte a seguir, iniciamos pela criação do formulário (**BootstrapFormBuilder**). Em seguida, são criados os elementos de seleção (**TRadioGroup**, **TCheckGroup**, **TCombo**, **TSelect**, **TMultiSearch**, etc). Posteriormente, abrimos uma transação (**TTransaction**) com a base de dados **samples** e carregamos em memória todos os objetos da classe **Category**, por meio do seu método **all()**. Percorremos um a um desses objetos (**\$collection**), adicionando em um vetor (**\$items**) indexado pelo código (**id**), tendo como valor o nome (**name**). Após adicionarmos todos os itens em um vetor (**\$items**), a transação com a base de dados é encerrada. Logo após, utilizamos o método **addItems()** para adicionar as opções aos elementos (**TRadioGroup**, **TCheckGroup**, **TCombo**, etc). Em seguida, os objetos são adicionados ao formulário e a ação **onSend()** é criada para o envio dos dados.

app/control/Presentation/Forms/FormDBManualSelectionView.class.php

```
<?php
class FormDBManualSelectionView extends TPage
{
    private $form;

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

        // cria o formulário
        $this->form = new BootstrapFormBuilder;
        $this->form->setFormTitle(_t('Manual database selections'));

        // cria os campos do formulário
        $radio   = new TRadioGroup('radio');
        $check   = new TCheckGroup('check');
        $combo   = new TCombo('combo');
        $select  = new TSelect('select');
        $search  = new TMultiSearch('search');
        $autocomp = new TEntry('autocomplete');

        // define algumas propriedades
        $search->setMinLength(1);
        $radio->setLayout('horizontal');
        $check->setLayout('horizontal');
```

```

// abre a transação
TTransaction::open('samples');

// carrega os objetos Category
$collection = Category::all();

// cria um vetor indexado
$items = array();
foreach ($collection as $object)
{
    $items[$object->id] = $object->name;
}
TTransaction::close();

// adiciona os itens aos objetos
$radio->addItems($items);
$check->addItems($items);
$combo->addItems($items);
$select->addItems($items);
$search->addItems($items);
$autocomp->setCompletion( array_values( $items ) );

// adiciona os campos ao formulário
$this->form->addField( [new TLabel('TRadioGroup:')], [$radio] );
$this->form->addField( [new TLabel('TCheckGroup:')], [$check] );
$this->form->addField( [new TLabel('TCombo:')], [$combo] );
$this->form->addField( [new TLabel('TSelect:')], [$select] );
$this->form->addField( [new TLabel('TMultiSearch (minlen 1):')], [$search] );
$this->form->addField( [new TLabel('Autocomplete:')], [$autocomp] );

$select->setSize('100%', 70);
$search->setSize('100%', 50);

// cria a ação de envio
$this->form->addAction('Send', new TAction(array($this, 'onSend')),
    'fa:check-circle-o green');

parent::add($this->form);
}

```

O método `onSend()`, por sua vez, coleta os itens selecionados no formulário por meio do método `getData()`, e os exibe por meio de um diálogo.

```

public function onSend($param)
{
    $data = $this->form->getData(); // obtém os dados do formulário

    // mantém o formulário preenchido após a postagem
    $this->form->setData($data);

    // cria uma string com os valores dos campos
    $message = 'Radio : ' . $data->radio . '<br>';
    $message.= 'Check : ' . print_r($data->check, TRUE) . '<br>';
    $message.= 'Combo : ' . $data->combo . '<br>';
    $message.= 'Select : ' . print_r($data->select, TRUE) . '<br>';
    $message.= 'MultiSearch: ' . print_r($data->search, TRUE) . '<br>';
    $message.= 'Autocomplete: ' . $data->autocomplete;

    // exibe a mensagem
    new TMessage('info', $message);
}

```

Obs: Esta abordagem, permite a transformação dos valores do BD antes de sua apresentação, caso necessário.

Na próxima figura, podemos ver o resultado da execução deste programa.

The screenshot shows a Delphi application window titled "Seletores manuais com dados". Inside, there are several components demonstrating manual selection:

- TRadioButton:** Three radio buttons labeled "Frequente", "Casual", and "Varejista".
- TCheckGroup:** Three checkboxes labeled "Frequente", "Casual", and "Varejista".
- TCombo:** A dropdown menu.
- TSelect:** A dropdown menu containing "Frequente", "Casual", and "Varejista".
- TMultiSearch (minlen 1):** A search input field showing "Frequente" and "Casual" as selected items.
- Autocomplete:** An input field.

At the bottom left is a "Send" button, and at the bottom right is the author's name "MARCOS ANTONIO RAFAEL DA FONSECA".

Figura 51 Formulário com seletores com valores do banco

4.5.15 Seleções automáticas

No exemplo anterior, vimos como utilizar valores de uma tabela do banco de dados em elementos de seleção. Apesar de simples, esta abordagem ainda pode ser otimizada. Pensando na agilidade da utilização deste tipo de elemento de seleção, foram criados novos componentes: (**TDBRadioGroup**, **TDBCheckGroup**, **TDBCombo**, **TDBSelect**, **TDBMultiSearch**, **TDBUniqueSearch**, e **TDBEntry**). O objetivo destes componentes é automatizar a seleção de valores a partir de registros existentes no banco de dados.

Neste exemplo, criamos uma página e dentro dela colocamos alguns estes componentes para seleção de registros do banco de dados (Todos começando com **TDB**). Estes componentes recebem os mesmos parâmetros: **nome** (nome do campo dentro do formulário); **banco** (banco de dados a ser usado para conexão); **modelo** (nome da classe Active Record a ser usada para buscar os registros); **chave** (nome do campo que será usado como chave do componente de seleção); **valor** (nome do campo que será exibido no componente de seleção); **ordem** (coluna usada para ordenação); e **critério** (objeto **TCriteria** usado para filtrar as opções exibidas). Conforme pode ser acompanhado no código-fonte a seguir, a exceção é o componente **TDBEntry**, que não recebe o parâmetro **chave**.

No momento da criação dos componentes (**TDBRadioGroup**, **TDBCheckGroup**, **TDBCombo**, **TDBSelect**, e etc), uma conexão já é realizada com a base de dados, baseada nos parâmetros informados. Neste momento, os valores já são carregados, preenchendo os elementos de seleção, conforme os parâmetros informados. A exceção são os

componentes `TDBUniqueSearch` e `TDBMultiSearch` que não pré-carregam os valores. Estes dois componentes somente buscam na base de dados quando o usuário começar a digitar. Isto é importante para ter desempenho ao buscar em tabelas grandes.

Em seguida, os elementos ainda são acrescentados ao formulário pelo método `addFields()`, e alguns valores default são definidos para os campos, por meio do método `setValue()`, e mais algumas propriedades são definidas, com destaque para os métodos `setLayout()`, que define a orientação das opções em radios e checks; `setUseButton()` que transforma radios e checks em botões; e `enableSearch()`, que habilita busca por caracteres em uma combo.

Os objetos `TMultisearch` ainda possuem algumas características especiais, definidas por métodos como `setMinLength()`, que define o tamanho mínimo em caracteres para habilitar a busca textual; `setMaxSize()`, que define a quantidade máxima de opções que poderão ser selecionadas; e `setMask()` que define a máscara de apresentação de resultados. É importante notar que inclusive atributos de objetos relacionados “`{state-name}`” podem ser usados na máscara, desde que os métodos de associação (Ex: `get_state()` neste caso) estejam definidos.

Por fim, uma ação para envio dos dados é criada por meio do método `addAction()`, que criará um botão vinculado ao método `onSend()`.

[app/control/Presentation/Forms/FormDBAutoSelectionView.class.php](#)

```
<?php
class FormDBAutoSelectionView extends TPage
{
    private $form;

    function __construct()
    {
        parent::__construct();
        $this->form = new BootstrapFormBuilder();
        $this->form->setFormTitle(_t('Automatic database selections'));
        $this->form->generateAria();

        // cria os campos (name, database, model, key, value, [order], [criteria])
        $radio      = new TDBRadioGroup('radio', 'samples', 'Category', 'id', 'name');
        $radio2     = new TDBRadioGroup('radio2','samples', 'Category', 'id', '{id} - {name}');
        $check      = new TDBCheckGroup('check', 'samples', 'Category', 'id', 'name');
        $check2     = new TDBCheckGroup('check2','samples', 'Category', 'id', '{id} - {name}');
        $combo      = new TDBCombo('combo', 'samples', 'Category', 'id', 'name');
        $combo2     = new TDBCombo('combo2', 'samples', 'Category', 'id', 'name');
        $select     = new TDBSelect('select', 'samples', 'Category', 'id', 'name');
        $search     = new TDBMultiSearch('search', 'samples', 'Category', 'id', 'name');
        $unique     = new TDBUniqueSearch('unique', 'samples', 'City', 'id', 'name');
        $autocomp   = new TDBEntry('autocomplete', 'samples', 'Category', 'name');

        // adiciona botão de criar novo registro após campo de busca (after $unique)
        $button = TButton::create('new', ['CityWindow', 'onClear'], '', 'fa:plus-circle green');
        $button->class = 'btn btn-default inline-button';
        $button->title = _t('New');
        $unique->after($button);
        $this->form->addField($button);
    }
}
```

```

// ajusta as proporções das colunas
$this->form->setColumnClasses(2, ['col-sm-4', 'col-sm-8']);

// adiciona os campos ao formulário
$this->form->addFields( [new TLabel('TDBRadioGroup:'), [$radio] ]);
$this->form->addFields( [new TLabel('TDBChekGroup:'), [$check] ]);
$this->form->addFields( [new TLabel('TDBRadioGroup (button):'), [$radio2] ]);
$this->form->addFields( [new TLabel('TDBChekGroup (button):'), [$check2] ]);
$this->form->addFields( [new TLabel('TDBCombo:'), [$combo] ]);
$this->form->addFields( [new TLabel('TDBCombo (search):'), [$combo2] ]);
$this->form->addFields( [new TLabel('TDBSelect:'), [$select] ]);
$this->form->addFields( [new TLabel('TDBMultiSearch:'), [$search] ]);
$this->form->addFields( [new TLabel('TDBUniqueSearch:'), [$unique] ]);
$this->form->addFields( [new TLabel('TDBEntry:'), [$autocomp] ]);

// opções específicas das buscas (unique e multi)
$search->setMinLength(1);
$unique->setMinLength(1);
$search->setMask('{name} ({id})');
$unique->setMask('{{id}} <b>{name}</b> - {state->name}');
$search->setOperator('like');

// define valores default
$radio->setValue(2);
$radio2->setValue(2);
$check->setValue(array(1,3));
$check2->setValue(array(1,3));
$combo->setValue(2);
$combo2->setValue(2);
$select->setValue(array(1,2));
$search->setValue(array(1,2));
$unique->setValue(2);

// define outras propriedades
$radio->setLayout('horizontal');
$check->setLayout('horizontal');
$radio2->setLayout('horizontal');
$check2->setLayout('horizontal');
$radio2->setUseButton();
$check2->setUseButton();
$combo->setSize('100%');
$combo2->setSize('100%');
$combo2->enableSearch();
$select->setSize('100%',100);
$search->setSize('100%',70);
$unique->setSize('100%');
$autocomp->setSize('100%');

// botão de ação
$this->form->addAction('Send', new TAction(array($this, 'onSend')),
    'fa:check-circle-o green');

parent::add($this->form);
}

```

Ao ser clicado, o botão de ação do formulário simplesmente apresenta os valores selecionados. Para tal, os dados do formulário são lidos pelo método `getData()` para o objeto `$data`. Então, a string `$message` é criada pela concatenação dos atributos deste objeto. Por fim, os valores selecionados são apresentados pela classe `TMessage`.

```

public function onSend($param)
{
    // obtém os dados do formulário
    $data = $this->form->getData();
}

```

```

// mantém o formulário preenchido após a postagem
$this->form->setData($data);

// cria uma string com os valores dos campos
$message = 'Radio : ' . $data->radio . '<br>';
$message.= 'Check : ' . print_r($data->check, TRUE) . '<br>';
$message.= 'Radio (button) : ' . $data->radio2 . '<br>';
$message.= 'Check (button) : ' . print_r($data->check2, TRUE) . '<br>';
$message.= 'Combo : ' . $data->combo . '<br>';
$message.= 'Combo2 : ' . $data->combo2 . '<br>';
$message.= 'Select : ' . print_r($data->select, TRUE) . '<br>';
$message.= 'Search : ' . print_r($data->search, TRUE) . '<br>';
$message.= 'Unique: ' . print_r($data->unique, TRUE) . '<br>';
$message.= 'Autocomplete: ' . $data->autocomplete;

// exibe a mensagem
new TMessage('info', $message);
}

}

```

Na próxima figura, podemos ver o resultado da execução deste programa.

The screenshot shows a web application interface for testing various form components. At the top, it says "Seletores automáticos" and "MARCOS ANTONIO RAFAEL DA FONSECA -". Below this, there are several input fields:

- TDBRadioGroup:** A radio button group with three options: Frequent (selected), Casual, and Varejista.
- TDBCheckGroup:** A checkbox group with three checkboxes: Frequent (selected), Casual, and Varejista.
- TDRadioButtonGroup (use button):** A radio button group with three buttons: 1 - Frequent, 2 - Casual, and 3 - Varejista.
- TDBCheckGroup (use button):** A checkbox group with three buttons: 1 - Frequent, 2 - Casual, and 3 - Varejista.
- TDBCombo:** A dropdown menu showing the option "Casual".
- TDBCombo (with search):** A dropdown menu showing the option "Casual".
- TDBSelect:** A dropdown menu showing the options "Frequente", "Casual", and "Varejista", with "Frequente" currently selected.
- TDBMultiSearch:** A search bar containing two items: "Frequente (1)" and "Casual (2)".
- TDBUniqueSearch:** A search bar containing the item "(2) Porto Alegre - RS".
- TDBEntry:** An empty text input field.

At the bottom left is a "Send" button, and at the bottom right is the text "MARCOS ANTONIO RAFAEL DA FONSECA -".

Figura 52 Formulário com seletores automáticos do banco

4.5.16 Interações dinâmicas

Até o momento, vimos vários exemplos de formulários e de componentes. Em todos os casos, somente tivemos acesso aos dados digitados pelo usuário após a postagem do formulário. Mas em muitas situações, precisamos ter acesso aos dados digitados em interações dinâmicas, como em um evento de saída de um campo ou na troca de um elemento de uma combo.

O Adianti Framework suporta interações dinâmicas em vários tipos de objetos. Elementos como: TEntry, TDate, TPassword, TText, e TSpinner, possuem um método chamado `setExitAction()`, que permite definir uma ação a ser executada quando o usuário retirar o foco do campo. A ação a ser executada deve ser obrigatoriamente um método estático (`public static function`).

Além destas, as classes TCombo, TCheckGroup, TRadioGroup, TUniqueSearch, TMultiSearch, TSortList, e TSelect e seus respectivos TDB's possuem o método `setChangeAction()`, que permite definir uma ação a ser executada quando o usuário alterar o valor selecionado. Em ambas as situações, a ação realizada na saída de um campo ou na alteração, poderá desencadear uma alteração imediata em um outro campo.

Este exemplo inicia com a criação de um formulário rápido (`BootstrapFormBuilder`), que terá basicamente cinco campos (`input_exit`, `response_a`, `combo_change`, `response_b` e `response_c`). Este formulário terá uma ação a ser executada quando o mesmo for postado, vinculada ao método `onView()`.

Sobre o campo `input_exit`, executaremos o método `setExitAction()`, que permite definir uma ação a ser executada quando o campo perder o foco. Neste caso, será executado o método `onExitAction()`. Já sobre o campo `combo_change`, executaremos o método `setChangeAction()`, que permite definir uma ação a ser executada quando o elemento da combo é alterado. Neste, será executado o método `onChangeAction()`.

app/control/Presentation/Forms/FormInteractionsView.class.php

```
<?php
class FormInteractionsView extends TPage
{
    private $form;
    function __construct()
    {
        parent::__construct();

        // cria um formulário
        $this->form = new BootstrapFormBuilder('form_interaction');
        $this->form->setFormTitle(_t('Dynamic interactions'));

        // cria alguns campos (TEntry e TCombo)
        $input_exit    = new TEntry('input_exit');
        $response_a   = new TEntry('response_a');
        $combo_change = new TCombo('combo_change');
        $response_b   = new TCombo('response_b');
        $response_c   = new TEntry('response_c');

        $response_a->setEditable(FALSE); // torna não-editável
        $response_c->setEditable(FALSE); // torna não-editável

        // adiciona alguns itens nas combos
        $combo_items = array();
        $combo_items['a'] = 'Item a';
        $combo_items['b'] = 'Item b';
        $combo_items['c'] = 'Item c';
        $combo_change->addItems($combo_items);
        $response_b->addItems($combo_items);
```

```

// adiciona os campos ao formulário
$this->form->addFields( [new TLabel('Input with exit action')], [$input_exit] );
$this->form->addFields( [new TLabel('Response A')], [$response_a] );
$this->form->addFields( [new TLabel('Combo change action')], [$combo_change] );
$this->form->addFields( [new TLabel('Response B')], [$response_b] );
$this->form->addFields( [new TLabel('Response C')], [$response_c] );

// ação default de envio dos dados
$this->form->addAction('View', new TAction(array($this, 'onView')),
    'fa:search');

// define a ação de saída para o campo input_exit
$exit_action = new TAction(array($this, 'onExitAction'));
$input_exit->setExitAction($exit_action);

// define a ação de alteração para a combo combo_change
$change_action = new TAction(array($this, 'onChangeAction'));
$combo_change->setChangeAction($change_action);

parent::add($this->form); // adiciona o formulário à página
}

```

Quando o usuário sair do campo `input_exit`, o método `onExitAction()` será executado. O objetivo deste método é alterar o valor de outros campos do formulário. Para tal, este método constrói um objeto (`$obj`) que contém em seus atributos, valores que preencherão os campos do formulário. O método `TForm::sendData()` é utilizado para alterar os valores do formulário por JavaScript. Neste caso, alteramos o valor do campo `response_a`, com base no que o usuário digitou no campo `input_exit`, e também com base na hora atual. E enviamos o valor “`a`”, para selecionar uma opção na `combo_change`. Como a combo também possui um evento de troca de valores vinculado, chamado `onChangeAction()`, este será executado indiretamente. Ao fim, também é exibida uma mensagem de sucesso ao usuário.

```

public static function onExitAction($param)
{
    $obj = new StdClass;
    $obj->response_a = 'Resp. for '.$param['input_exit'].' at ' . date('H:m:s');
    $obj->combo_change = 'a';

    // envia os dados ao formulário
    TForm::sendData('form_interaction', $obj);
    new TMessage('info', 'Message on field exit. <br>You have typed: ' .
        $param['input_exit']);
}

```

Quando o usuário alterar o valor selecionado da combo `combo_change`, o método `onChangeAction()` será executado. O objetivo deste método também é alterar outros campos do formulário. Inicialmente é alterado o campo `response_c`. Para tal, é construído um objeto (`$obj`) no qual cada atributo representa um campo do formulário. Então utilizamos o método `sendData()` da classe `TForm` para enviar os dados dinamicamente ao formulário. Outra operação realizada neste exemplo é a recarga da combo. A combo é recarregada com novas opções, por meio do seu método estático `reload()`. Em ambos os casos, é imprescindível identificar o nome do formulário corretamente no primeiro parâmetro.

```

public static function onChangeAction($param)
{
    $obj = new StdClass;
    $obj->response_c = 'Resp. for opt "'.$param['combo_change'].'" .
        "' . date('H:m:s');
    TForm::sendData('form_interaction', $obj); // envia os dados ao formulário

    $options = array();
    $options[1] = $param['combo_change'] . ' - one';
    $options[2] = $param['combo_change'] . ' - two';
    $options[3] = $param['combo_change'] . ' - three';
    // recarrega as opções da combo
    TCombo::reload('form_interaction', 'response_b', $options);
}

```

Quando o usuário enviar os dados do formulário por meio da ação “View”, será executado o método `onView()`. Neste método, devemos manter o formulário preenchido, mesmo havendo um campo, cujo preenchimento depende de seu campo anterior. O campo “`response_b`” depende do conteúdo de “`combo_change`”, que é dinâmico. Neste caso, para manter o formulário preenchido mesmo em campos dinâmicos, devemos utilizar o método `sendData()` para enviar os dados do formulário por meio de uma requisição JavaScript mesmo após a postagem. Esta técnica permite que o evento de troca da combo “`combo_change`” seja disparado novamente, sendo que logo em seguida, o campo “`response_b`” também tem seu valor definido de maneira síncrona. Os demais campos do formulário, são preenchidos pelo método `setData()`.

```

public function onView()
{
    $data = $this->form->getData(); // lê os dados do formulário

    // monta um objeto para enviar dados após o post
    $obj = new StdClass;
    $obj->combo_change = $data->combo_change; // dispara a ação de mudança
    $obj->response_b = $data->response_b; // atualiza o valor após mudança

    TForm::sendData('form_interaction', $obj); // envia dados dinamicamente
    $this->form->setData($data); // mantém o formulário preenchido

    new TMessage('info', str_replace(',', ', <br> ', json_encode($data)));
}
}

```

Na próxima figura, podemos ver o resultado da execução deste programa.

The screenshot shows a web page titled "Interações dinâmicas" with a sub-header "MARCOS ANTONIO RAFAEL DA FONSECA". The form contains several input fields:

- "Input with exit action": A text input containing "123".
- "Response A": A text input containing "Resp. for 123 at 23:07:44".
- "Combo with change action": A dropdown menu currently showing "Item b".
- "Response B": A dropdown menu currently showing "b - two".
- "Response C": A text input containing "Resp. for opt 'b' 23:07:46".

At the bottom left is a "View" button, and at the bottom right is the name "MARCOS ANTONIO RAFAEL DA FONSECA".

Figura 53 Formulário com interações dinâmicas entre campos

4.5.17 Habilitando e desabilitando campos

No exemplo anterior, pudemos ver como configurar interações dinâmicas em formulários, por meio dos métodos `setExitAction()`, que permite definir uma ação para o evento de saída de um campo e `setChangeAction()` que permite definir uma ação para o evento de troca de valores.

Neste exemplo, serão demonstrados mais alguns recursos que dão dinamicidade a um formulário: habilitação, desabilitação e limpeza de campos. Para demonstrar tais recursos, utilizaremos a ação de troca de valor de um radio button, por meio do método `setChangeAction()`. Conforme a opção clicada na radio button, um grupo de campos será habilitado.

Como podemos ver neste exemplo, no método construtor definimos uma interface com vários campos (`TRadioGroup`, `TCombo`, `TEntry`, etc.). Estes campos estão dentro de um formulário (`BootstrapFormBuilder`). O primeiro campo do formulário (`radio_enable`) trata-se de uma combo, cujo evento de saída (`setChangeAction`) está configurado para executar o método `onChangeRadio()`.

[app/control/Presentation/Forms/FormEnableDisableView.class.php](#)

```
<?php
class FormEnableDisableView extends TPage
{
    private $form;
    function __construct()
    {
        parent::__construct();

        // cria o formulário
        $this->form = new BootstrapFormBuilder('form_enable_disable');
        $this->form->setFormTitle(_t('Enable/disable interactions'));

        // cria os campos
        $radio_enable = new TRadioGroup('enable');
        $radio_enable->addItems(array('1'=>'Enable group 1', '2'=>'Enable group 2'));
        $radio_enable->setLayout('horizontal');
        $radio_enable->setValue(1);

        $block1_combo = new TCombo('block1_combo');
        $block1_entry = new TEntry('block1_entry');
        $block1_spinner = new TSpinner('block1_spinner');
        $block2_date = new TDate('block2_date');
        $block2_entry = new TEntry('block2_entry');
        $block2_check = new TCheckGroup('block2_check');

        // adiciona os itens
        $block1_combo->addItems(array(1=>'One', 2=>'Two'));
        $block1_spinner->setRange(1,100,10);
        $block2_check->addItems(array('Y'=>'Yes', 'N'=>'No'));

        // adiciona os campos ao formulário
        $this->form->addField([], [$radio_enable] );
        $this->form->addField([new TLabel('group #1')], [$block1_combo] );
        $this->form->addField([], [$block1_entry] );
        $this->form->addField([], [$block1_spinner] );
        $this->form->addField([new TLabel('group #2')], [$block2_date] );
        $this->form->addField([], [$block2_entry] );
        $this->form->addField([], [$block2_check] );
```

```

$radio_enable->setChangeAction( new TAction( array($this, 'onChangeRadio')) );
self::onChangeRadio( array('enable'=>1) );

$ vbox->add($this->form);
}

```

O que o método `onChangeRadio()` faz é verificar em qual elemento do radio button o usuário clicou. Caso seja na primeira opção (1), habilita o primeiro grupo de campos, por meio do método `enableField()` de cada classe, e desabilita o segundo grupo de campos por meio do método `disableField()`, além de limpá-los por meio do método `clearField()`. No ELSE, são realizadas as operações inversas. É importante notar que tanto o método `enableField()`, quanto `disableField()` e `clearField()` recebem por parâmetro o nome do formulário (definido no construtor da `BootstrapFormBuilder`), e o nome de cada campo.

Obs: É importante executar o método específico de cada classe, tendo em vista que sua implementação varia de uma classe para outra.

```

public static function onChangeRadio($param)
{
    if ($param['enable'] == 1)
    {
        TCombo::enableField('form_enable_disable', 'block1_combo');
        TEntry::enableField('form_enable_disable', 'block1_entry');
        TSpinner::enableField('form_enable_disable', 'block1_spinner');

        TDate::disableField('form_enable_disable', 'block2_date');
        TEntry::disableField('form_enable_disable', 'block2_entry');
        TCheckGroup::disableField('form_enable_disable', 'block2_check');

        TDate::clearField('form_enable_disable', 'block2_date');
        TEntry::clearField('form_enable_disable', 'block2_entry');
        TCheckGroup::clearField('form_enable_disable', 'block2_check');
    }

    else
    {
        TCombo::disableField('form_enable_disable', 'block1_combo');
        TEntry::disableField('form_enable_disable', 'block1_entry');
        TSpinner::disableField('form_enable_disable', 'block1_spinner');

        TDate::enableField('form_enable_disable', 'block2_date');
        TEntry::enableField('form_enable_disable', 'block2_entry');
        TCheckGroup::enableField('form_enable_disable', 'block2_check');

        TCombo::clearField('form_enable_disable', 'block1_combo');
        TEntry::clearField('form_enable_disable', 'block1_entry');
        TSpinner::clearField('form_enable_disable', 'block1_spinner');
    }
}

```

Na próxima figura, podemos ver o resultado da execução deste programa.

The screenshot shows a web-based configuration interface. At the top, it says "Interações habilita/desabilita" and "MARCOS ANTONIO RAFAEL DA FONSECA -". Below this, there are two groups of input fields. Group #1 contains a radio button group ("Enable group 1" and "Enable group 2"), a dropdown menu, a date input field with a calendar icon, and a large text area. Group #2 contains a checkbox group ("Yes" and "No"). A watermark "MARCOS ANTONIO RAFAEL DA FONSECA -" is located in the bottom right corner of the form area.

Figura 54 Habilitando e desabilitando campos

4.5.18 Botão de busca de registros

Em exemplos anteriores, estudamos diferentes maneiras de utilizar componentes de seleção de registros. Alguns desses, como a **TDBCombo**, podem ser utilizados sempre que a quantidade de registros a ser apresentada em tela não é muito grande, visto que todos registros são carregados para a tela de uma única vez. Outros, como a **TDBUniqueSearch** e **TDBMultiSearch** permitem a busca conforme a digitação ocorre, minimizando a comunicação com o servidor. Quando tivermos uma quantidade muito grande de registros para o usuário selecionar (Ex: pessoas) e desejarmos estabelecer critérios mais complexos de busca (vários filtros), podemos utilizar os componentes especializados em busca de registros (**TSeekButton** e **TDBSeekButton**), que abrem uma janela de buscas, onde o usuário poderá filtrar e selecionar um registro.

O componente **TSeekButton** permite realizar a seleção de registros a partir de uma janela de buscas criada manualmente na qual podemos utilizar diversos campos de busca. Já o componente **TDBSeekButton** cria uma janela de buscas padrão com campos código e nome, uma abordagem mais simplificada que será vista neste exemplo.

Para demonstrar o componente de busca de registros, criaremos um formulário para localização de cidades, produtos e clientes. Cada campo terá um botão de busca que abrirá uma janela e permitirá ao usuário selecionar um, dentre vários registros, sendo que o selecionado retornará para o formulário antes da postagem. É importante notar que a janela que irá se abrir possui paginação. Desta forma, em momento algum carregaremos em memória todos os registros da base de dados.

Conforme pode ser visto no código-fonte a seguir, este exemplo inicia pela criação formulário (**BootstrapFormBuilder**). Após a criação do formulário, os objetos de localização são criados (**\$city_id**, **\$product_id**, e **\$customer_id**).

Todos objetos recebem como parâmetro: nome do campo no formulário, nome da base de dados, nome do formulário para o qual os dados serão enviados, classe Active Record para buscas, e nome do campo descritivo usado nas buscas. Para cada um destes campos, criaremos outro correspondente para receber a descrição (`$city_name`, `$product_name`, e `$customer_name`). Os campos são vinculados por meio do método `setAuxiliar()`.

O segundo campo de buscas (`$product_id`), recebe um filtro de dados, por meio do método `setCriteria()`, que permite determinar que a janela de buscas já abrirá filtrada. O terceiro campo de buscas (`$customer_id`), possui uma máscara de exibição para o campo descritivo, o que é determinado pelo método `setDisplayMask()`.

Após configurar os campos, estes são adicionados ao formulário pelo método `addFields()`. Outras características são configuradas como o tamanho do campo código, por meio do método `setSize()` e o bloqueio da edição do campo nome por meio do método `setEditable()`.

app/control/Presentation/Forms/FormSeekButtonView.class.php

```
<?php
class FormSeekButtonView extends TPage
{
    private $form;

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

        // cria o formulário
        $this->form = new BootstrapFormBuilder('form_seek');
        $this->form->setFormTitle(_t('Seek button'));

        // campo para busca de cidades
        $city_id = new TDBSeekButton('city_id', 'samples', 'form_seek', 'City', 'name');
        $city_name = new TEntry('city_name');
        $city_id->setAuxiliar($city_name);

        // filtro para aplicar em produtos
        $criteria = new TCriteria;
        $criteria->add(new TFilter('id', '>=', 1));
        $criteria->add(new TFilter('id', '<=', 10));
        $criteria->setProperty('order', 'id');

        // campo para busca de produtos
        $product_id = new TDBSeekButton('product_id', 'samples', 'form_seek', 'Product',
        'description');
        $product_name = new TEntry('product_name');
        $product_id->setCriteria($criteria);
        $product_id->setAuxiliar($product_name);

        // campo para busca de clientes
        $customer_id = new TDBSeekButton('customer_id', 'samples', 'form_seek',
        'Customer', 'name');
        $customer_name = new TEntry('customer_name');
        $customer_id->setDisplayMask('{name} - {city->name} - {city->state->name}');
        $customer_id->setDisplayLabel('Informações do cliente');
        $customer_id->setAuxiliar($customer_name);
```

```

$city_name->setEditable(FALSE);
$product_name->setEditable(FALSE);
$customer_name->setEditable(FALSE);

// ajusta as proporções dos slots do formulário
$this->form->setColumnClasses(2, ['col-sm-3', 'col-sm-9']);

// adiciona os campos ao formulário
$this->form->addField([new TLabel('Standard Seek')], [$city_id]);
$this->form->addField([new TLabel('Standard with filter')], [$product_id]);
$this->form->addField([new TLabel('Standard with mask')], [$customer_id]);

$city_id->setSize(80);
$product_id->setSize(80);
$customer_id->setSize(80);
$city_name->setSize('calc(100% - 120px)');
$product_name->setSize('calc(100% - 120px)');
$customer_name->setSize('calc(100% - 120px)');
$city_name->style .= ';margin-left:3px';
$product_name->style .= ';margin-left:3px';
$customer_name->style .= ';margin-left:3px';

// adiciona um botão de ação
$this->form->addAction('Save', new TAction(array($this, 'onSave')), 'fa:save');

parent::add($this->form);
}

```

O formulário possui ainda um botão de ação para exibir os valores informados após a postagem. Ao ser clicado, o botão executa o método `onSave()`, que obtém os dados do formulário por meio do método `getData()`, e monta uma string, a fim de exibir os valores do formulário por meio de um diálogo apresentado pela classe `TMessage`.

```

public function onSave($param)
{
    $data = $this->form->getData(); // obtém os dados do formulário
    $this->form->setData($data); // mantém o formulário preenchido

    // cria a mensagem
    $message = 'City id : ' . $data->city_id . '<br>';
    $message.= 'Product id : ' . $data->product_id . '<br>';
    $message.= 'Customer id : ' . $data->customer_id . '<br>';

    // exibe a mensagem
    new TMessage('info', $message);
}
}

```

Na próxima figura, podemos ver o resultado da execução deste programa.

Figura 55 Formulário com botões de busca de registros

4.5.19 Edição de HTML

Frequentemente nas aplicações precisamos de um campo para entrada de dados com maior riqueza na formatação. Como exemplos, poderíamos citar a digitação de mensagens de e-mail, a digitação de notícias em um site, a configuração de mensagens personalizadas para serem exibidas ao usuário, ou até mesmo um campo descritivo no qual desejamos oferecer mais recursos de entrada ao usuário.

Pensando nisso, foi criado um componente para edição de HTML, o `THtmlEditor`. Este componente cria uma área de edição de HTML para o usuário. Este componente pode ser acrescentado em um formulário como qualquer outro componente de entrada de dados, e devolve ao usuário os dados digitados no formato HTML.

Para demonstrar a utilização deste componente, vamos criar uma página contendo um formulário simples, somente com um `THtmlEditor` e um botão de ação, que ao ser clicado, dispara o método `onShow()`. O papel do método `onShow()` é simplesmente exibir um diálogo de mensagem contendo a mensagem digitada pelo usuário.

app/control/Presentation/Forms/FormHtmlEditorView.class.php

```
<?php
class FormHtmlEditorView extends TPage
{
    private $form;

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

        // cria o formulário
        $this->form = new BootstrapFormBuilder;
        $this->form->setFormTitle( _t('Html Editor') );

        // cria os elementos de input
        $html = new THtmlEditor('html_text');
        $html->setSize( '100%', 200 );
        $html->setOption('placeholder', 'type here...');

        // adiciona os objetos ao formulário
        $this->form->addFields( [$html] );

        // cria uma ação
        $this->form->addAction('Show', new TAction(array($this, 'onShow')),
            'fa:check-circle-o green');

        parent::add($this->form);
    }

    public function onShow($param)
    {
        $data = $this->form->getData(); // obtém os dados
        $this->form->setData($data);
        new TMessage('info', $data->html_text); // exibe uma mensagem
    }
}
```

Na próxima figura, podemos ver o resultado da execução deste programa.

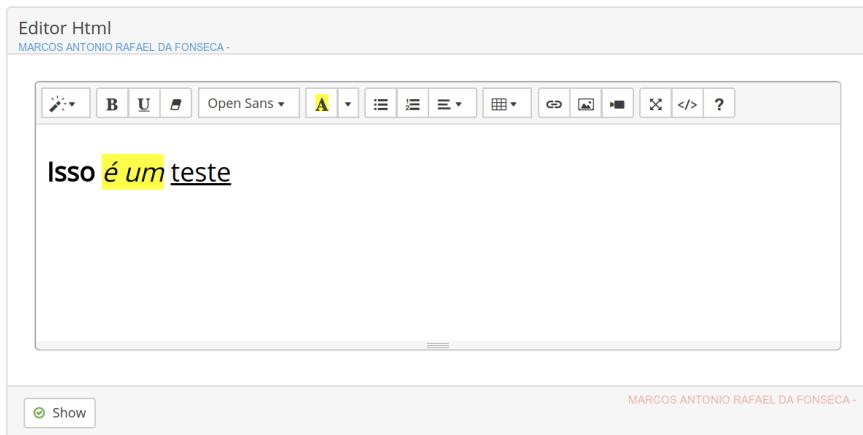


Figura 56 Componente de edição de HTML

4.5.20 Listas de ordenação

Neste exemplo, será demonstrado o componente `TSortList`, que cria uma lista, que permite ao usuário definir a ordem de seus elementos utilizando o recurso de clicar e arrastar. Além de permitir ordenar elementos dentro de uma mesma lista, o componente `TSortList` permite também arrastar elementos entre diferentes listas de ordenação, permitindo o usuário “escolher” elementos de uma lista e “jogar” para outra.

Neste exemplo, criamos um formulário com dois elementos `TSortList`. Para adicionarmos elementos é utilizado o método `addItems()`, da mesma forma que já foi visto em classes como `TCombo`, `TRadioGroup` e outras. É passado um vetor como parâmetro, onde os índices identificam cada uma das opções. O método `setSize()` permite definir o tamanho da lista, e o método `connectTo()` permite conectar uma lista à outra. Quando o método `connectTo()` é executado, o usuário pode arrastar elementos entre diferentes listas.

Além de conectar listas, podemos programar uma ação para ser executada sempre que o usuário mover um elemento de uma lista. Isto é permitido pelo método `setChangeAction()`, que vincula uma ação (objeto `TAction`). Neste exemplo, programamos o método `onChangeAction()` para ser executado sempre que um elemento da primeira lista for movimentado. Este método somente apresentará as opções escolhidas em tela por meio da classe `TMessage`.

Este formulário ainda possui um botão de ação, que executa o método `onSend()`, que por sua vez exibe em tela o resultado de cada uma das listas. O conteúdo resultante de cada campo é um vetor contendo os índices das opções selecionadas e ordenado conforme a seleção do usuário.

app/control/Presentation/Forms/FormSortView.class.php

```
<?php
class FormSortView extends TPage
{
    private $form;

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

        // cria o formulário
        $this->form = new BootstrapFormBuilder;
        $this->form->setFormTitle(_t('Sort List'));

        // cria as listas de ordenação
        $list1 = new TSortList('list1');
        $list2 = new TSortList('list2');

        // adiciona os itens em cada lista
        $list1->addItems( array('1' => 'One', '2' => 'Two', '3' => 'Three') );
        $list2->addItems( array('a' => 'A', 'b' => 'B', 'c' => 'C') );

        $list1->setSize(200, 100);
        $list2->setSize(200, 100);

        // conecta as listas
        $list1->connectTo($list2);
        $list2->connectTo($list1);

        // define evento de troca de valor
        $list1->setChangeAction(new TAction(array($this, 'onChangeAction')));

        // adiciona as listas ao formulário
        $this->form->addField([$list1, $list2]);

        // adiciona a ação ao formulário
        $this->form->addAction('Send', new TAction(array($this, 'onSend')),
            'fa:check-circle-o');

        parent::add($this->form);
    }

    public static function onChangeAction($param)
    {
        new TMessage('info', 'Change action<br>'.
            'List1: ' . implode(',', $param['list1']) . '<br>' .
            'List2: ' . implode(',', $param['list2']));
    }

    public function onSend($param)
    {
        $data = $this->form->getData(); // obtém os dados do formulário
        $this->form->setData($data); // mantém o formulário preenchido

        // cria uma string com os valores dos campos
        $message = 'List 1: ' . implode(',', $data->list1) . '<br>';
        $message.= 'List 2 : ' . implode(',', $data->list2) . '<br>';

        new TMessage('info', $message); // exibe a mensagem
    }
}
```

Na próxima figura, podemos ver o resultado da execução deste programa.

The screenshot shows a user interface titled "Lista de ordenação". It displays two separate lists. The first list, on the left, has three items: "One", "Two", and "Three", each in its own row. The second list, on the right, has three items: "A", "B", and "C", also each in their own row. Below these lists is a button labeled "Send". The entire interface is contained within a light gray box with a thin border. At the top of the box, there is some small text: "Listas de ordenação" and "MARCOS ANTONIO RAFAEL DA FONSECA -". At the bottom right of the box, there is another instance of the same text: "MARCOS ANTONIO RAFAEL DA FONSECA -".

Figura 57 Componente para ordenação de elementos

Neste exemplo, criamos duas listas de ordenação com conteúdos estáticos (vetores). Porém, também podemos utilizar o componente `TDBSortList`, que cria uma lista de ordenação a partir de registros do banco de dados. Sua sintaxe é similar aos demais componentes que iniciam por `TDB`, e ele recebe em seu construtor: o nome do campo no formulário, o nome da base de dados, o nome da classe Active Record, a chave, a descrição do campo, e a ordenação. O campo de valor aceita máscaras, como exibido no exemplo a seguir. O próximo exemplo demonstra uma lista de seleção de clientes conectada a outra lista vazia.

```
app/control/Presentation/Forms/FormDBSortView.class.php
class FormDBSortView extends TPage
{
    private $form;

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

        $this->form = new BootstrapFormBuilder;
        $this->form->setFormTitle(_t('Sort DB List'));

        $list1 = new TDBSortList('list1', 'samples', 'Customer', 'id', '{name} (#{id})',
'id');
        $list2 = new TSortList('list2');

        $list1->setSize(200, 200);
        $list2->setSize(200, 200);

        $list1->connectTo($list2);
        $list2->connectTo($list1);

        // adiciona as listas ao formulário
        $this->form->addFields([$list1, $list2]);

        // adiciona ação
        $this->form->addAction( 'Send', new TAction(array($this, 'onSend')),
'fa:check-circle-o');

        parent::add($this->form);
    }
}
```

O método `onSend()` é acionado pelo botão de ação do formulário e trata da exibição dos valores de ambas as listas.

```
public function onSend($param)
{
    // obtém os dados
    $data = $this->form->getData();

    // mantém o formulário preenchido
    $this->form->setData($data);

    // cria uma string com os valores selecionados
    $message = 'List 1: ' . implode(',', $data->list1) . '<br>';
    $message.= 'List 2 : ' . implode(',', $data->list2) . '<br>';

    // exibe a mensagem
    new TMessage('info', $message);
}
```

A figura a seguir demonstra o formulário criado.

Figura 58 Componente para ordenação de elementos de uma tabela

4.5.21 Formulários MVC reutilizáveis

Nos exemplos anteriores, você deve ter percebido que o visual da página foi construído inteiramente no método construtor da classe, sendo que o comportamento da página estava definido por outros métodos da mesma classe (`onSend`, `onSave`). Idealmente, no padrão MVC, o aspecto visual (aparência), deve estar separado do aspecto de controle (comportamento). Com a separação do aspecto visual em relação ao aspecto de controle, podemos inclusive utilizar a mesma interface (view), em diferentes páginas (control), com comportamentos diferentes em cada caso.

O propósito do próximo exemplo é demonstrar a separação total entre os aspectos de apresentação (view) e controle (control), visto que o aspecto modelo (model) já está separadamente representado pelas classes Active Record.

Para iniciar, vamos criar uma classe que representa um formulário. Esta classe será filha de `BootstrapFormBuilder` e já trará os campos do formulário, sua organização visual, algumas combos preenchidas. Esta classe ficará armazenada em `app/view`, diretório onde ficarão as classes que representam interfaces (views) reutilizáveis.

Como a classe `BootstrapFormBuilder` será a superclasse, sempre que precisarmos executar um de seus métodos, como `setFormTitle()`, basta precedermos a chamada com “`parent::`”. Dentro do método construtor, criamos os objetos do formulário, adicionamos eles ao formulário, pelo método `addFields()`. Veja que aqui não definimos a ação do formulário, pois a mesma será definida pela classe de controle.

`app/view/FormReusableView.class.php`

```
<?php
class FormReusableView extends BootstrapFormBuilder
{
    public function __construct()
    {
        parent::__construct();
        parent::setFormTitle('Form reusable');

        // cria os campos
        $id      = new TEntry('id');
        $name   = new TEntry('name');
        $address = new TEntry('address');
        $phone   = new TEntry('phone');
        $email   = new TEntry('email');
        $gender  = new TCombo('gender');
        $status  = new TCombo('status');

        // adiciona os campos ao formulário
        parent::addFields( [new TLabel('Id')], [$id] );
        parent::addFields( [new TLabel('Name')], [$name] );
        parent::addFields( [new TLabel('Address')], [$address] );
        parent::addFields( [new TLabel('Phone')], [$phone] );
        parent::addFields( [new TLabel('Email')], [$email] );
        parent::addFields( [new TLabel('Gender')], [$gender] );
        parent::addFields( [new TLabel('Status')], [$status] );

        $gender->addItems( [ 'M' => 'Male', 'F' => 'Female' ] );
        $status->addItems( [ 'S' => 'Single', 'C' => 'Committed', 'M' => 'Married' ] );
    }
}
```

A classe criada representa o aspecto visual de um formulário. O próximo passo, é utilizá-la a partir de uma classe de controle. Como vimos, uma classe de apresentação (view) pode ser utilizada por diferentes classes de controle (control).

O exemplo a seguir busca demonstrar como criar uma classe de controle que utiliza o formulário definido anteriormente. Para tal, no método construtor, instanciamos a classe de apresentação (`FormReusableView`), e acrescentamos uma ação a ele por meio do método `addAction()`, que não existe na classe `FormReusableView`, mas existe na sua superclasse (`BootstrapFormBuilder`). Ao final do construtor, adicionamos este formulário completo à página.

app/control/Presentation/Extras/FormReusableControl.class.php

```
<?php
class FormReusableControl extends TPage
{
    private $form;

    function __construct()
    {
        parent::__construct();
        $this->form = new FormReusableView; // instancia a view

        // adiciona a ação
        $this->form->addAction('Show', new TAction(array($this, 'onShow')),
            'fa:check-circle-o');
        parent::add($this->form);
    }

    public function onShow($param)
    {
        $data = $this->form->getData();
        $this->form->setData($data);
        new TMessage('info', str_replace(',', ', <br>', json_encode($data)));
    }
}
```

Na próxima figura, podemos ver o resultado da execução deste programa.

The screenshot shows a web-based form interface. At the top left, it says "Form reusable". Below that, there is a small blue link "MARCOS ANTONIO RAFAEL DA FONSECA -". The form itself has seven input fields: "Id" (text), "Name" (text), "Address" (text), "Phone" (text), "Email" (text), "Gender" (dropdown menu), and "Status" (dropdown menu). At the bottom left of the form area, there is a button labeled ">Show". At the bottom right, there is another small blue link "MARCOS ANTONIO RAFAEL DA FONSECA -".

Figura 59 Formulário MVC reutilizável

4.6 Datagrids

Ao longo do livro, vimos como criar páginas simples, como organizar o visual da aplicação por meio de containers, como interagir com o usuário por meio de diálogos, e também como coletar informações por meio de formulários. O próximo passo está em apresentar as informações ao usuário no formato de datagrids, permitindo que o mesmo possa interagir sobre os dados tomando ações. Umadatagrid geralmente apresenta uma coleção de Active Records vindos da base de dados. Entretanto, antes de interagirmos com a base de dados, vamos aprender os conceitos básicos sobre datagrids, trabalhando com dados estáticos. Trabalharemos integrando os componentes visuais e o banco de dados no próximo capítulo.

4.6.1 Datagrids

O objetivo deste exemplo é construir uma datagrid para apresentação de alguns dados estáticos, e apresentar ao usuário duas possíveis ações para cada registro.

Conforme pode ser acompanhado no código-fonte a seguir, a criação da datagrid inicia pela criação de um objeto `TDataGrid`. Neste exemplo, também estamos habilitando o Popover (balão quando passa o mouse sobre a linha). Então, são criadas quatro colunas (`TGridColumn`) para serem acrescentadas à datagrid. Cada `TGridColumn` recebe em seu construtor: o atributo do objeto de dados que será exibido na datagrid; o título da coluna; o alinhamento da coluna; e a largura da coluna em pixels ou percentual. Após, as colunas são adicionadas à datagrid por meio do método `addColumn()`.

Ao adicionarmos as colunas à datagrid pelo método `addColumn()`, podemos opcionalmente passar como parâmetro uma ação (objeto `TAction`). Este objeto define a ação de clique sobre o título da coluna. Esta ação normalmente é utilizada para operações de ordenação e não tem nenhuma relação com a ação de linha, usada normalmente para operações como editar e excluir. Ao declararmos a ação, podemos opcionalmente definir quais parâmetros ela vai receber, por meio de um vetor (2º parâmetro do construtor).

Sobre cada coluna, ainda podemos definir propriedades. Neste caso, estamos definindo o título (`title`) de cada coluna, que é exibido quando passamos o mouse sobre.

Após adicionarmos as colunas à datagrid, duas ações são criadas: `onView()` e `onDelete()`. Cada ação é um objeto `TDataGridAction`. O objeto da classe `TDataGridAction` recebe como parâmetro o método a ser executado, e em segundo lugar um vetor com os parâmetros que serão passados para a ação quando a mesma for executada. Perceba que ao passarmos `['code'=>'{code}']` estamos na verdade definindo que a ação receberá na posição “`code`” o atributo `{code}` do objeto clicado. Podemos passar como parâmetro qualquer atributo do objeto, bastando mapear neste vetor.

Após criarmos as ações, estas são acrescentadas à datagrid por meio do `addAction()`, definindo label e ícone. Além disso, a datagrid é criada em memória pelo método `createModel()` e a datagrid é adicionada à página.

`app/control/Presentation/Datagrid/DatagridCustomView.class.php`

```
<?php
class DatagridCustomView extends TPage
{
    private $datagrid;
    public function __construct()
    {
        parent::__construct();
        $this->datagrid = new TDataGrid; // cria a datagrid
        $this->datagrid->enablePopover('Details', '<b>Code:</b> {code} ...');
        $code      = new TGridColumn('code',   'Code',   'center', '10%');
        $name     = new TGridColumn('name',   'Name',   'left',   '30%');
        $address  = new TGridColumn('address', 'Address', 'left',   '30%');
        $telephone= new TGridColumn('fone',   'Phone',  'left',   '30%');
```

```

// adiciona as colunas àdatagrid, e define ação de clique no header
$this->datagrid->addColumn($code, new TAction([$this, 'onColumnAction']),
    ['column' => 'code']) );
$this->datagrid->addColumn($name, new TAction([$this, 'onColumnAction']),
    ['column' => 'name']) );
$this->datagrid->addColumn($city, new TAction([$this, 'onColumnAction']),
    ['column' => 'city']) );
$this->datagrid->addColumn($state,new TAction([$this, 'onColumnAction']),
    ['column' => 'state']) );

$code->title = 'Here is the code';
$name->title = 'Here is the name';

// cria duas ações de linha
$action1 = new TDataGridAction([$this, 'onView'], ['code'=>'{code}',
    'name' => '{name}']);
$action2 = new TDataGridAction([$this, 'onDelete'], ['code'=>'{code}']);

// adiciona as ações nadatagrid
$this->datagrid->addAction($action1, 'View', 'fa:search blue');
$this->datagrid->addAction($action2, 'Delete', 'fa:trash red');

$this->datagrid->createModel(); // cria o modelo dadatagrid
parent::add($this->datagrid); // acrescenta adatagrid à página
}

```

Logo após criar adatagrid no método construtor, sobrecrevemos o método `show()`. Assim, sempre que a página for carregada, o método `onReload()` também será executado para inserir alguns objetos nadatagrid.

```

function show()
{
    $this->onReload();
    parent::show();
}

```

O método `onReload()` acionado sempre antes do `show()`, será responsável por acrescentar alguns objetos àdatagrid, o que ocorre pelo método `addItem()`. O método `addItem()` recebe um objeto em que cada um de seus atributos devem corresponder aos nomes identificados no momento da criação das colunas (`code`, `name`, etc).

```

function onReload()
{
    $this->datagrid->clear();

    // acrescenta um objeto àdatagrid
$item = new StdClass;
$item->code    = '1';
$item->name   = 'Aretha Franklin';
$item->city   = 'Memphis';
$item->state  = 'Tennessee (US)';
$this->datagrid->addItem($item);

$item = new StdClass;
$item->code    = '2';
$item->name   = 'Eric Clapton';
$item->city   = 'Ripley';
$item->state  = 'Surrey (UK)';
$this->datagrid->addItem($item);

// ...
}

```

Em seguida, temos o método a ser executado quando o usuário clicar sobre o título da coluna. Neste caso, estamos fazendo a leitura do atributo `column`, que identifica a coluna clicada e exibindo este nome em tela por meio de um diálogo. O nome do atributo utilizado aqui é `column`, pois assim ele foi definido no mapeamento da ação.

```
public function onColumnAction($param)
{
    // obtém o nome da coluna e exibe a mensagem.
    $column = $param['column'];
    new TMessage('info', "You clicked at the column <b>{$column}</b>");
}
```

Por fim, temos os dois métodos que respondem às ações de linha dadatagrid: `onView()` e `onDelete()`. Estes métodos recebem por padrão um vetor (`$param`), que contém o valor do atributo passado como parâmetro (definido no construtor da `TDataGridViewAction`). Neste exemplo, ambas as ações têm apenas como função exibir as informações do atributo passado como parâmetro.

```
public function onView($param)
{
    // obtém os atributos e exibe por um diálogo
    $code = $param['code'];
    $name = $param['name'];
    new TMessage('info', "The code is: <b>$code</b> The name is : <b>$name</b>");
}
```

Observe que o segundo método é estático. Quando um método não for estático e ele for executado em uma ação, toda a classe é processada (método construtor, método requisitado, método show). Quando um método for estático, somente este método será executado. Porém, dentro de um método estático, não é possível utilizar a variável `$this`, uma vez que não existe objeto instanciado, pois a execução não passou pelo construtor. O tamanho da requisição HTTP de um método estático é portanto bem menor.

```
public static function onDelete($param)
{
    // get the parameter and shows the message
    $code = $param['code'];
    new TMessage('error', "The register <b>{$code}</b> may not be deleted");
}
```

Na próxima figura, podemos ver o resultado da execução deste programa.

Code	Name	Address	Phone
1	Fábio Locatelli	Rua Expedicionario	1111-1111
2	Julia Haubert	Rua Expedicionarios	2222-2222
3	Carlos Ranzi	Rua Oliveira	3333-3333
4	Daline DallOglio	Rua Oliveira	4444-4444

Figura 60 Datagrids

4.6.2 Datagrids Bootstrap

Você deve ter percebido que as datagrids possuem um visual peculiar no Adianti Framework. Porém, é possível modificá-lo de duas maneiras: a primeira é por meio de CSS, uma vez que a estrutura do componente é montada por HTML e CSS; e a segunda maneira é utilizando um Decorator. Um Decorator permite adicionar funcionalidades a uma classe existente em tempo de execução, diminuindo a necessidade de alterarmos a classe original, e também a necessidade de estendê-la.

O Adianti Framework possui decorators que são utilizados para “modificar” os componentes nativos e deixá-los com uma aparência diferente. No exemplo a seguir, utilizamos o `BootstrapDatagridWrapper`, que é um decorator que modifica umadatagrid nativa do Framework e transforma-a em umadatagrid com a aparência Bootstrap. Basta instanciarmos a classe `BootstrapDatagridWrapper` e passarmos instância de `TDataGrid` como parâmetro. O restante do código-fonte foi suprimido por ser similar ao anterior, mas você encontra ele integralmente nos fontes do Tutor.

Além de deixarmos adatagrid com aparência Bootstrap, realizamos outras alterações neste exemplo. Uma delas é exibir as ações dadatagrid no formato de botão. Para tal, basta executarmos o método `setUseButton(TRUE)` sobre o objeto `TDataGridAction`. Além disso, também criamos um painel ao redor dadatagrid (`TPanelGroup`). Estes elementos estão em destaque no código-fonte a seguir.

app/control/Presentation/Datagrid/DatagridBootstrapView.class.php

```
<?php
class DatagridBootstrapView extends TPage
{
    private $datagrid;

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

        // cria adatagrid
        $this->datagrid = new BootstrapDatagridWrapper(new TDataGrid);
        $this->datagrid->enablePopover('Details', '<b>Code:</b> {code} ...');

        // cria as colunas dadatagrid
        $code   = new TDataGridColumn('code',    'Code',    'center', '10%');
        $name  = new TDataGridColumn('name',    'Name',    'left',   '30%');
        $city   = new TDataGridColumn('city',    'City',    'left',   '30%');
        $state  = new TDataGridColumn('state',   'State',   'left',   '30%');

        // adiciona as colunas nadatagrid
        $this->datagrid->addColumn($code);
        $this->datagrid->addColumn($name);
        $this->datagrid->addColumn($city);
        $this->datagrid->addColumn($state);

        // cria as ações
        $action1 = new TDataGridAction([$this, 'onView'],  ['code'=>'{code}', 
                                                               'name' => '{name}']);
        $action2 = new TDataGridAction([$this, 'onDelete'], ['code'=>'{code}']);
    }
}
```

```

// habilita a aparência da ação como botão
$action1->setUseButton(TRUE);
$action2->setUseButton(TRUE);

// adiciona a ação àdatagrid.
$this->datagrid->addAction($action1, 'View', 'fa:search blue');
$this->datagrid->addAction($action2, 'Delete', 'fa:trash red');

// cria a estrutura dadatagrid em memória
$this->datagrid->createModel();

// cria um painel ao redor dadatagrid, habilitando scroll horizontal
$panel = new TPanelGroup('Bootstrap Datagrid');

// adiciona adatagrid ao panel, habilitando scroll horizontal
$panel->add($this->datagrid)->style = 'overflow-x:auto';

// footer do painel
$panel->addFooter('footer');

parent::add($panel);
}

public function onReload()
{
    // ...
}

public function onDelete($param)
{
    // ...
}

public function onView($param)
{
    // ...
}

public function show()
{
    // ...
}
}

```

Na próxima figura, podemos ver o resultado da execução deste programa.

Datagrid Bootstrap					
MARCOS ANTONIO RAFAEL DA FONSECA -					
	Code	Name	City	State	
Q View	Delete	1	Aretha Franklin	Memphis	Tennessee (US)
Q View	Delete	2	Eric Clapton	Ripley	Surrey (UK)
Q View	Delete	3	B.B. King	Itta Bena	Mississippi (US)
Q View	Delete	4	Janis Joplin	Port Arthur	Texas (US)
footer					

Figura 61 Datagrid Bootstrap

4.6.3 Datagrids com grupos de ações

Em exemplos anteriores, vimos como criar datagrids com ações. Na maioria das vezes, umadatagrid terá 2 ou 3 ações, não muito mais. Entretanto, nas vezes em que for necessário criar umadatagrid com um número maior de ações, a interface ficará poluída com muitos ícones. Nestes casos, é indicado o uso do agrupamento de ações, que é possível por meio da classe `TDataGridActionGroup`, que será demonstrada nesse exemplo. Esta classe cria um menu Dropdown, e encapsula as ações dentro deste menu, possibilitando a disponibilização de um número maior de opções ao usuário.

Neste exemplo, criaremos umadatagrid convencional, da mesma forma que nos exemplos anteriores. Estadatagrid terá algumas colunas, instanciadas por meio da classe `TDataGridColumn`. Adatagrid terá também algumas ações (`TDataGridAction`), que por sua vez não serão adicionadas diretamente nadatagrid. Para adicionar as ações nadatagrid, utilizamos uma classe intermediária que é `TDataGridActionGroup`, que por sua vez disponibiliza métodos como `addHeader()`, `addAction()` e `addSeparator()`, que permitem respectivamente: adicionar um cabeçalho, uma ação, e um separador. O agrupamento de ações cria um único botão de ação para cada linha dadatagrid, que expande quando clicado, apresentando as opções disponíveis.

app/control/Presentation/Datagrid/DatagridActionGroupView.class.php

```
<?php
class DatagridActionGroupView extends TPage
{
    private $datagrid;
    public function __construct()
    {
        parent::__construct();
        $this->datagrid = new BootstrapDatagridWrapper(new TDataGrid);

        // cria as colunas
        $code      = new TDataGridColumn('code',     'Code',     'right',  '10%');
        $name     = new TDataGridColumn('name',     'Name',     'left',   '30%');
        $address  = new TDataGridColumn('address',  'Address',  'left',   '30%');
        $telephone = new TDataGridColumn('fone',    'Phone',    'left',   '30%');

        $this->datagrid->addColumn($code);
        $this->datagrid->addColumn($name);
        $this->datagrid->addColumn($address);
        $this->datagrid->addColumn($telephone);

        // cria as ações dadatagrid
        $action1 = new TDataGridAction([$this, 'onView'],      ['code' => '{code}',           'name' => '{name}']);
        $action2 = new TDataGridAction([$this, 'onDelete'],    ['code' => '{code}']);
        $action3 = new TDataGridAction([$this, 'onViewCity'],  ['city' => '{city}']);

        // define labels é ícones usando a biblioteca Font Awesome
        $action1->setLabel('View name');
        $action1->setImage('fa:search #7C93CF');
        $action2->setLabel('Try to delete');
        $action2->setImage('fa:trash red');
        $action3->setLabel('View city');
        $action3->setImage('fa:hand-pointer-o green');
```

```

// cria o agrupamento de ações
$action_group = new TDataGridActionGroup('Actions', 'fa:th');

// adiciona as ações ao agrupamento
$action_group->addHeader('Available Options');
$action_group->addAction($action1);
$action_group->addAction($action2);
$action_group->addSeparator();
$action_group->addHeader('Another Options');
$action_group->addAction($action3);

$this->datagrid->addActionGroup($action_group); // adiciona o agrupamento

$this->datagrid->createModel(); // cria o modelo

$panel = TPanelGroup::pack(_t('Datagrid'), $this->datagrid, 'footer');
parent::add($panel);
}

// demais métodos . . .
}

```

Na próxima figura, podemos ver o resultado da execução deste programa.

	Code	Name	Address	Phone
Actions ▾	1	Fábio Locatelli	Rua Expedicionario	1111-1111
Available Options		Julia Haubert	Rua Expedicionarios	2222-2222
Q View name		Carlos Ranzi	Rua Oliveira	3333-3333
✗ Try to delete		Daline DallOglio	Rua Oliveira	4444-4444
Another Options				
📍 View address				

Figura 62 Datagrids com grupos de ações

4.6.4 Datagrids com ações condicionais

Quando criamos uma datagrid, geralmente desejamos exibir todas as suas ações ao usuário. Mas, em alguns casos algumas ações devem ser suprimidas. Em algumas situações, por exemplo, não desejamos exibir o botão de excluir apenas para alguns registros em que essa ação não deve ser executada.

Para permitir esconder a ação do usuário em alguns casos, foi criado o método `setDisplayCondition()` da classe `TDataGridAction`. Este método permite definir um método do usuário como parâmetro. O método definido pelo usuário, que neste caso é `displayColumn()`, receberá o objeto que representa a linha corrente (Active Record) e deve retornar um valor booleano (TRUE/FALSE) se a ação correspondente àquela linha, deve ou não ser exibida.

Neste exemplo, o método `displayColumn()`, recebe o objeto, e avalia se o código está compreendido entre 1 e 4, retornando TRUE neste caso, e FALSE, caso contrário. Desta maneira, somente os objetos cujos códigos são 2 ou 3 possuem a respectiva ação exibida, o que pode ser conferido na figura a seguir.

Datagrid com ações condicionais			
MARCOS ANTONIO-RAFAEL DA FONSECA -			
Code	Name	Address	Phone
1	Fábio Locatelli	Rua Expedicionario	1111-1111
2	Julia Haubert	Rua Expedicionarios	2222-2222
3	Carlos Ranzl	Rua Oliveira	3333-3333
4	Daline DallOglia	Rua Oliveira	4444-4444

footer

MARCOS ANTONIO-RAFAEL DA FONSECA -

Figura 63 Datagrid com ações condicionais

app/control/Presentation/Datagrid/DatagridConditionalActionView.class.php

```
<?php
class DatagridConditionalActionView extends TPage
{
    private $datagrid;

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

        // cria adatagrid
        $this->datagrid = new BootstrapDatagridWrapper(new TDataGrid);

        // cria as colunas
        $code      = new TDataGridColumn('code',     'Code',     'right',  '10%');
        $name     = new TDataGridColumn('name',     'Name',     'left',   '30%');
        $address  = new TDataGridColumn('address',  'Address',  'left',   '30%');
        $telephone = new TDataGridColumn('fone',     'Phone',    'left',   '30%');

        // adiciona as colunas
        $this->datagrid->addColumn($code);
        $this->datagrid->addColumn($name);
        $this->datagrid->addColumn($address);
        $this->datagrid->addColumn($telephone);

        // cria a ação
        $action1 = new TDataGridAction(array($this, 'onView'), ['name' => '{name}'] );

        // aqui define a condição de exibição da ação
        $action1->setDisplayCondition( array($this, 'displayColumn') );

        // adiciona ação na datagrid.
        $this->datagrid->addAction($action1, 'View', 'fa:search blue');

        $this->datagrid->createModel(); // cria o modelo

        parent::__add($this->datagrid);
    }
}
```

```

// Define quando a ação será exibida
public function displayColumn( $object )
{
    if ($object->code > 1 AND $object->code < 4) {
        return TRUE;
    }
    return FALSE;
}

// outros métodos
function onReload() {}
function onView($param) {}
function show()
{
    $this->onReload();
    parent::show();
}

```

Obs: Neste exemplo, realizamos uma verificação bastante simples. Mas você poderá comparar campos de status (ativo/cancelado), datas (passado, futuro), ou valores (positivo, negativo) para determinar se uma ação deverá ou não ser exibida.

4.6.5 Datagrids com Popover

Na maioria das vezes, uma datagrid é suficiente para apresentar em tela, os dados de uma tabela. Entretanto, em algumas situações precisamos mostrar uma quantidade grande de informações, e o tamanho de tela pode ser um fator limitante em relação às colunas. Nestes casos, podemos utilizar o Popover, que é um recurso da biblioteca Bootstrap, que permite criar uma camada de informação ao passarmos o mouse sobre um elemento. Já vimos como usar o método `enablePopover()`, um método simples da classe `TDataGrid` para definição de um popover. Neste exemplo, vamos abordar uma maneira diferente para definir popovers.

Neste exemplo, criaremos um Popover para cada linha dadatagrid no método `onReload()`. Para usar o Popover, inicialmente criamos umadatagrid da maneira convencional, e acrescentamos colunas a ela, o que é realizado no método construtor.

O efeito de Popover é habilitado no método `onReload()`, que por sua vez, adiciona as linhas àdatagrid por meio do método `addItem()`. O método `addItem()` retorna a linha acrescentada (objeto `TTableRow`). A partir deste objeto em memória (`$row`), ligamos o Popover por meio da definição do atributo `popover` para '`true`', bem como definimos a posição do popover (opcionalmente) por meio do atributo `popside` para `top`, `bottom`, `left` ou `right`. O título e o conteúdo do Popover são definidos por meio dos atributos `poptitle` e `popcontent`.

Obs: Ao utilizarmos a classe `TDataGrid`, podemos executar seu método `enablePopover()`, que define um Popover global para toda adatagrid, baseado em uma string definida pelo usuário, podendo conter variáveis.

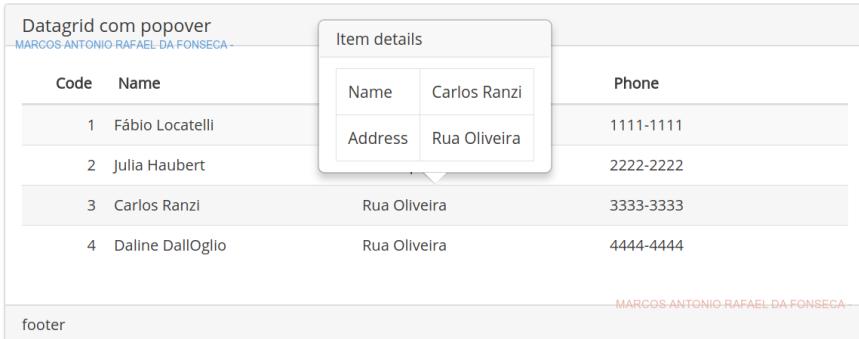


Figura 64 Datagrids com Popover

app/control/Presentation/Datagrid/DatagridPopView.class.php

```
<?php
class DatagridPopView extends TPage
{
    private $datagrid;
    public function __construct()
    {
        parent::__construct();

        // cria a datagrid
        $this->datagrid = new BootstrapDatagridWrapper(new TDataGrid);
        $this->datagrid->style = 'width: 100%';

        // cria as colunas
        $this->datagrid->addColumn( new TDataGridColumn('code', 'Code', 'center' ) );
        $this->datagrid->addColumn( new TDataGridColumn('name', 'Name', 'left' ) );
        $this->datagrid->addColumn( new TDataGridColumn('city', 'City', 'left' ) );
        $this->datagrid->addColumn( new TDataGridColumn('state', 'State', 'left' ) );

        $this->datagrid->createModel();
        parent::add($this->datagrid);
    }

    function onReload()
    {
        $this->datagrid->clear();

        // adiciona um objeto
        $item = new StdClass;
        $item->code      = '1';
        $item->name      = 'Fábio Locatelli';
        $item->address   = 'Rua Expedicionario';
        $item->fone      = '1111-1111';

        // adiciona a linha
        $row = $this->datagrid->addItem($item);

        // habilita um popover ao elemento
        $row->popover = 'true'; // habilita popover
        $row->popside = 'top'; // posição do popover (top, bottom, left, right)
        $row->popcontent = "<table class='popover-table'>
                            <tr><td>Name</td><td>{$item->name}</td></tr>...</table>";
        $row->poptitle = 'Item details';
    }

    // ...
}
```

4.6.6 Datagrids com rolagem horizontal

Vimos que em algumas situações, é importante exibirmos um conjunto maior de informações em uma linha de datagrid. Uma forma de resolver este problema, é criando um Popover, outra maneira é ativando a rolagem horizontal, que será vista neste exemplo. A rolagem horizontal, cria uma barra de scroll ao redor dadatagrid no sentido horizontal de rolagem. Neste caso, as colunas não mudam de tamanho a medida em que redimensionamos a janela, somente a área de rolagem que diminui.

Neste exemplo, criamos umadatagrid, por meio da classe `BootstrapDatagridWrapper`, em seguida definimos a largura dadatagrid. Esta largura é importante, pois define um tamanho “confortável” para o usuário visualizar todas as informações de uma linha. Ao redor dessadatagrid, as barras de rolagem serão criadas. Após, adicionamos algumas colunas pelo método `addColumn()`. Adatagrid deste exemplo terá somente uma ação, acrescentada pelo método `addAction()`.

Adatagrid é empacotada dentro de um painel (`TPanelGroup`). Um dos segredos desta abordagem (além de definir a largura em `1900px`), é alterar o estilo do corpo do painel que conterá adatagrid para `"overflow-x:auto;"`, ativando a rolagem horizontal ao redor dadatagrid. O restante do exemplo é muito similar aos anteriores.

```
app/control/Presentation/Datagrid/DatagridHScrollView.class.php


---


class DatagridHScrollView extends TPage
{
    private $datagrid;

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

        // cria adatagrid
        $this->datagrid = new BootstrapDatagridWrapper(new TDataGrid);
        $this->datagrid->style = 'min-width: 1900px'; // define largura mínima fixa

        // adiciona as colunas
        $this->datagrid->addColumn( new TGridColumn('code', 'Code', 'center') );
        $this->datagrid->addColumn( new TGridColumn('name', 'Name', 'left') );
        $this->datagrid->addColumn( new TGridColumn('phone', 'Phone', 'left') );
        $this->datagrid->addColumn( new TGridColumn('email', 'Email', 'left') );
        $this->datagrid->addColumn( new TGridColumn('city', 'City', 'left') );
        $this->datagrid->addColumn( new TGridColumn('state', 'State', 'left') );
        // ...

        // cria a ação passando os atributos {code} e {name} como parâmetro
        $action1 = new TDataGridAction([$this, 'onView'], ['code'=>'{code}', 'name' => '{name}']);
        $this->datagrid->addAction($action1, 'View', 'fa:search blue');

        // cria a estrutura dadatagrid em memória
        $this->datagrid->createModel();

        // cria um painel ao redor dadatagrid
        $panel = new TPanelGroup(_t('Horizontal Scrollable Datagrids'));
        $panel->add($this->datagrid);
        $panel->addFooter('footer');
    }
}
```

```
// liga a rolagem horizontal ao redor da datagrid
$panel->getBody()->style = "overflow-x:auto;";
parent::add($panel);
}
// ...
}
```

Na figura a seguir, podemos conferir o resultado da execução deste programa.

Datagrid com scroll horizontal					
MARCOS ANTONIO RAFAEL DA FONSECA -					
Code	Name	Address	Phone	E	
Q 1	Fábio Locatelli	Rua Expedicionario	1111-1111	fa	
Q 2	Julia Haubert	Rua Expedicionarios	2222-2222	ju	
Q 3	Carlos Ranzi	Rua Oliveira	3333-3333	ca	
Q 4	Daline DallOglio	Rua Oliveira	4444-4444	da	

← →

MARCOS ANTONIO RAFAEL DA FONSECA -

footer

Figura 65 Datagrid com rolagem horizontal

4.6.7 Datagrids com rolagem vertical

Adatagrid padrão do Framework não possui rolagem, pois adapta a sua altura conforme a quantidade de elementos. Além disso, utiliza-se paginação quando temos muitos elementos de uma tabela do banco de dados para exibir. Mas isso não significa que uma datagrid não possa ter uma rolagem (scrolling). Para habilitar a rolagem em uma datagrid é bastante simples, e veremos como fazer a seguir.

Inicialmente vamos criar uma datagrid comum, por meio da classe `TDataGrid`. E logo em seguida, vamos utilizar dois métodos que farão toda a diferença: `makeScrollable()`, que tornará a rolagem na datagrid, e `setHeight()`, que determinará a altura fixa da datagrid. A utilização destes dois métodos, em conjunto, acrescentará barras de rolagem verticais na datagrid, mantendo o seu cabeçalho (colunas) fixo.

Logo em seguida, adicionamos quatro colunas (`code`, `name`, `address` e `phone`) à datagrid. Ao final, acrescentamos a datagrid à página.

app/control/Presentation/Datagrid/DatagridScrollView.class.php

```
<?php
class DatagridScrollView extends TPage
{
    private $datagrid;
    public function __construct()
    {
        parent::__construct();
        $this->datagrid = new BootstrapDatagridWrapper(new TDataGrid);

        // habilita a rolagem e define a altura
        $this->datagrid->makeScrollable();
        $this->datagrid->setHeight(300);
    }
}
```

```

// cria as colunas da datagrid
$code      = new TDataGridColumn('code',      'Code',      'right',    70);
$name      = new TDataGridColumn('name',      'Name',      'left',     180);
$address   = new TDataGridColumn('address',   'Address',   'left',     180);
$telephone = new TDataGridColumn('fone',      'Phone',     'left',     160);

// adiciona as colunas nadatagrid
$this->datagrid->addColumn($code);
$this->datagrid->addColumn($name);
$this->datagrid->addColumn($address);
$this->datagrid->addColumn($telephone);

$this->datagrid->createModel(); // cria o modelo (estrutura) dadatagrid
parent::add($this->datagrid); // adiciona adatagrid à página
}

```

O método `onReload()` será responsável por preencher adatagrid com objetos. Para criar o efeito de rolagem, precisaremos acrescentar vários objetos. Para tal, dentro de um laço de repetição, acrescentaremos 40 objetos com informações padronizadas.

```

function onReload()
{
    $this->datagrid->clear();

    for ($n=1; $n<=40; $n++)
    {
        $item = new StdClass;
        $item->code    = $n;
        $item->name    = 'Person name';
        $item->address  = 'Person address';
        $item->fone    = '1111-1111';
        $this->datagrid->addItem($item);
    }
}

// ...
}

```

Na figura a seguir podemos conferir o resultado da execução deste programa.

Datagrid com scroll vertical			
MAROS ANTONIO RAFAEL DA FONSECA -			
Code	Name	Address	Phone
1	Person name	Person address	1111-1111
2	Person name	Person address	1111-1111
3	Person name	Person address	1111-1111
4	Person name	Person address	1111-1111
5	Person name	Person address	1111-1111
6	Person name	Person address	1111-1111
7	Person name	Person address	1111-1111
8	Person name	Person address	1111-1111
9	Person name	Person address	1111-1111

MAROS ANTONIO RAFAEL DA FONSECA -

footer

Figura 66 Datagrid com rolagem vertical

4.6.8 Datagrid com colunas escondidas

Em exemplos anteriores, vimos como usar o Popover e a rolagem horizontal para exibir mais informações em umadatagrid do que a capacidade da resolução da tela. Outra forma muito interessante de contornar o problema da falta de espaço é a utilização de colunas auto escondidas. Este recurso permite definir uma resolução mínima de tela (largura) para a qual a coluna é exibida. Se a resolução de tela for inferior ao número informado, a coluna é escondida automaticamente. O Framework permite utilizar este recurso em tamanhos múltiplos de 100px (100, 200, 300, etc).

Para habilitar o recurso de auto esconder uma determinada coluna, basta obtermos o objeto que representa a coluna (**TGridColumn**). Neste caso, obtemos a partir do retorno do método **addColumn()**. Sobre o objeto que representa a coluna, executamos o método **enableAutoHide()**, informando uma quantidade de pixels na forma de um número múltiplo de 100. Quando a resolução da tela for inferior ao tamanho informado, a coluna será automaticamente escondida. Assim, em telas pequenas, somente as primeiras colunas serão exibidas. Já em telas maiores, mais colunas aparecerão. Dessa forma, é recomendado exibir nas primeiras colunas as informações mais importantes.

app/control/Presentation/Datagrid/DatagridHidableView.class.php

```
class DatagridHidableView extends TPage
{
    private $datagrid;

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

        // cria a datagrid
        $this->datagrid = new BootstrapDatagridWrapper(new TDataGrid);
        $this->datagrid->width = '100%';

        // cria as colunas
        $this->datagrid->addColumn( new TGridColumn('code', 'Code', 'center') );
        $this->datagrid->addColumn( new TGridColumn('name', 'Name', 'left') );

        // adiciona as colunas, e obtém o retorno
        $col_birthdate = $this->datagrid->addColumn(new TGridColumn('birthdate'...));
        $col_phone     = $this->datagrid->addColumn(new TGridColumn('phone',...));

        $col_email     = $this->datagrid->addColumn( ... );
        $col_city      = $this->datagrid->addColumn( ... );
        $col_state     = $this->datagrid->addColumn( ... );
        $col_country   = $this->datagrid->addColumn( ... );

        // liga o auto esconder, definindo o tamanho para tal
        $col_birthdate->enableAutoHide(500);
        $col_phone->enableAutoHide(600);
        $col_email->enableAutoHide(700);
        $col_city->enableAutoHide(800);
        $col_state->enableAutoHide(900);
        $col_country->enableAutoHide(1000);
    }
}
```

```

// adiciona ações
$action1 = new TDataGridAction([$this, 'onView'], ['code'=>'{code}', 'name' => '{name}']);
$this->datagrid->addAction($action1, 'View', 'fa:search blue');

$this->datagrid->createModel();

$panel = new TPanelGroup(_t('Datagrid with hiddable columns') );
$panel->add($this->datagrid);
$panel->addFooter('footer');

parent::add($panel);
}

function onReload()
{
    // ...
}

public function show()
{
    $this->onReload();
    parent::show();
}
}

```

A próxima figura demonstra o resultado da execução deste programa. Infelizmente, por meio de uma imagem não é possível demonstrar este recurso. Mas se você redimensionar a tela durante a execução, perceberá algumas colunas sendo automaticamente escondidas.

Datagrid com colunas escondidas							
MARCOS ANTONIO RAFAEL DA FONSECA -							
Code	Name	Birthdate	Phone	Email	City	State	Country
Q 1	Aretha Franklin	25/03/1942	08 18 1235 1412 1231	aretha@email.com	Memphis	Tennessee	US
Q 2	Eric Clapton	30/03/1945	08 18 1235 1412 7476	eric@email.com	Ripley	Surrey	UK
Q 3	B.B. King	16/09/1925	08 18 1235 1412 6574	king@email.com	Itta Bena	Mississippi	US
Q 4	Janis Joplin	19/01/1943	08 18 1235 1412 6584	janis@email.com	Port Arthur	Texas	US

MARCOS ANTONIO RAFAEL DA FONSECA -

footer

Figura 67 Datagrid com colunas escondidas

4.6.9 Datagrid com datatables

Outra forma de exibir mais informações em umadatagrid do que a resolução horizontal da tela comporta, é utilizando a biblioteca datatables. A biblioteca datatables, usa o efeito de auto esconder as colunas para resoluções pequenas. Cada coluna escondida poderá ser vista por meio de um botão (+) criado no início de cada linha, que expande a linha verticalmente, exibindo as informações que não couberam naquela resolução.

Para habilitar a biblioteca datatables em umadatagrid, basta criar um atributo **datatable = 'true'** no objeto dedatagrid, o que pode ser visto em destaque a seguir.

```
app/control/Presentation/Datagrid/DatagridDatatableView.class.php


---


class DatagridDatatableView extends TPage
{
    private $datagrid;

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

        // cria adatagrid
        $this->datagrid = new BootstrapDatagridWrapper(new TDataGrid);
        $this->datagrid->width = '100%';
        $this->datagrid->datatable = 'true'; // liga Datatables

        // adiciona as colunas
        $this->datagrid->addColumn( new TDataGridColumn('code',      'Code',      'center') );
        $this->datagrid->addColumn( new TDataGridColumn('name',      'Name',      'left') );
        $this->datagrid->addColumn( new TDataGridColumn('address',   'Address',   'left') );
        $this->datagrid->addColumn( new TDataGridColumn('phone',     'Phone',     'left') );
        $this->datagrid->addColumn( new TDataGridColumn('city',      'City',      'left') );
        $this->datagrid->addColumn( new TDataGridColumn('status',    'Status',    'left') );
        $this->datagrid->addColumn( new TDataGridColumn('email',     'Email',     'left') );
        $this->datagrid->addColumn( new TDataGridColumn('gender',    'Gender',    'left') );
        $this->datagrid->addColumn( new TDataGridColumn('other1',   'Other1',   'left') );
        $this->datagrid->addColumn( new TDataGridColumn('other2',   'Other2',   'left') );
        $this->datagrid->addColumn( new TDataGridColumn('other3',   'Other3',   'left') );
        // ...

        // adiciona as ações
        $action1 = new TDataGridAction([$this, 'onView'], ['code'=>'{code}', 'name' => '{name}']);
        $this->datagrid->addAction($action1, 'View', 'fa:search blue');

        // cria a estrutura em memória
        $this->datagrid->createModel();

        $panel = new TPanelGroup('Datagrid datatable');
        $panel->add($this->datagrid);
        $panel->addFooter('footer');

        parent::__add($panel);
    }
    // ...
}
```

A figura a seguir demonstra umadatagrid com datatables. Veja que as colunas escondidas são exibidas por meio do botão (+) presente no início da linha.

Datagrid datatable											
MARCOS ANTONIO RAFAEL DA FONSECA -											
Code	Name	Address	Phone	Birthdate	City	Status	Email	Gender	Other1	Other2	
1	My friend nr. 1	Street	1111-1111	12/12/1990	New York	Married	friend1@test.com	male	other1	other2	(+)
Other3: other3											
Other4: other4											
Other5: other5											
Other6: other6											
Other7: other7											
Other8: other8											
Other9: other9											
Other10: other10											
2	My friend nr. 2	Street	2222-2222	12/12/1990	New York	Married	friend2@test.com	female	other1	other2	(+)
MARCOS ANTONIO RAFAEL DA FONSECA -											
footer											

Figura 68 Datagrid com datatables

4.6.10 Datagrid com busca rápida

Seguidamente precisamos buscar registros em umadatagrid. Normalmente esta operação envolve uma consulta no banco de dados, pois o cenário mais comum é utilizarmos adatagrid para exibir uma busca paginada (exibir apenas alguns registros de muitos).

Porém, em alguns casos específicos nos quais exibimos todos os registros disponíveis em tela, é útil podermos realizar uma busca em tela (somente dos registros visíveis). Este exemplo demonstra como criar um campo de busca para localizarmos no conteúdo dadatagrid visível em tela.

Para tal, a criação dadatagrid deste exemplo não é muito diferente dos demais. As mudanças iniciam-se com a criação do campo de busca (`$input_search`). Trata-se de um input normal, onde o usuário digitará o conteúdo a ser buscado. Este campo será posicionado no topo dadatagrid. Ele será adicionado no header do painel, por meio do método `addHeaderWidget()` da classe `TPanelGroup`.

Para habilitar a busca, é necessário executar o método `enableSearch()` dadatagrid. Este método recebe o input de buscas (`$input_search`) e também uma lista de atributos sobre os quais a busca ocorrerá. Podemos definir vários atributos usando a vírgula.

Neste exemplo a busca se dará sobre os campos `code`, `name`, `city` e `state`. Antes de ser adicionada em tela, adatagrid é posicionada dentro de um painel (`TPanelGroup`).

app/control/Presentation/Datagrid/DatagridSearchView.class.php

```

class DatagridSearchView extends TPage
{
    private $datagrid;

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

        // cria a datagrid
        $this->datagrid = new BootstrapDatagridWrapper(new Tdatagrid);
        $this->datagrid->width = '100%';

        // cria as colunas
        $this->datagrid->addColumn( new TdatagridColumn('code', 'Code', 'center' ) );
        $this->datagrid->addColumn( new TdatagridColumn('name', 'Name', 'left' ) );
        $this->datagrid->addColumn( new TdatagridColumn('city', 'City', 'left' ) );
        $this->datagrid->addColumn( new TdatagridColumn('state', 'State', 'left' ) );

        // cria as ações
        $action1 = new TdatagridAction([$this, 'onView'], ['code'=>'{code}', 'name' => '{name}']);
        $this->datagrid->addAction($action1, 'View', 'fa:search blue');

        // cria a estrutura
        $this->datagrid->createModel();

        // cria o campo de busca
        $input_search = new TEntry('input_search');
        $input_search->placeholder = _t('Search');
        $input_search->setSize('100%');

        // habilita busca nadatagrid
        $this->datagrid->enableSearch($input_search, 'code, name, city, state');

        // cria um painel ao redor dadatagrid
        $panel = new TPanelGroup(_t('Datagrid search'));
        $panel->addHeaderWidget($input_search);
        $panel->add($this->datagrid)->style = 'overflow-x:auto';
        $panel->addFooter('footer');

        parent::add($panel);
    }
}

```

Os demais métodos foram propositadamente suprimidos por realizarem ações que já abordamos anteriormente.

```

public function onReload()
{
    // ...
}

public function show()
{
    $this->onReload();
    parent::show();
}
}

```

A figura a seguir demonstra adatagrid com o campo de buscas no topo.

Busca em datagrid			
MARCOS ANTONIO RAFAEL DA FONSECA -			
Code	Name	City	State
1	Aretha Franklin	Memphis	Tennessee (US)
2	Eric Clapton	Ripley	Surrey (UK)
3	B.B. King	Itta Bena	Mississippi (US)
4	Janis Joplin	Port Arthur	Texas (US)

MARCOS ANTONIO RAFAEL DA FONSECA -

footer

Figura 69 Datagrid com busca rápida

4.6.11 Datagrid com formatação de colunas

Ao exibirmos dados em datagrids, é muito comum a necessidade de transformar (formatar) os dados antes da sua exibição. Algumas das transformações mais comuns são: formatar valores, datas, alterar as cores e estilo (negrito, itálico, sublinhado), exibir imagens, exibir elementos HTML alternativos, dentre outros.

A transformação em umadatagrid pode ser realizada de duas maneiras: logo antes dos objetos serem adicionados nadatagrid pelo método `addItem()` dadatagrid, ou por uma função transformadora. O uso de uma função transformadora é recomendado, uma vez que torna explícita a transformação. Além disso, uma função de transformação pode ser reaproveitada em diferentes datagrids.

Para demonstrar tal recurso, criamos umadatagrid (`TDataGridView`) e adicionamos algumas colunas. Em seguida, executamos o método `setTransformer()` sobre a coluna `$stock`. Este método define um outro método a ser executado sobre esta coluna dadatagrid. O objetivo do método de transformação (`formatSalary` neste caso) é receber o valor da coluna e retorná-la transformada. Ao final do construtor, adicionamos as colunas àdatagrid, criamos seu modelo e adicionamos adatagrid à página. O método de transformação não é executado neste instante, somente programado.

Neste exemplo também demonstramos o uso do método `setDataProperty()` responsável por definir uma propriedade de uma coluna. Neste caso, estamos alterando o estilo da coluna `stock` para negrito (`bold`).

app/control/Presentation/Datagrid/DatagridTransformView.class.php

```
<?php
class DatagridTransformView extends TPage
{
    private $datagrid;
    public function __construct()
    {
        parent::__construct();
```

```

// cria adatagrid
$this->datagrid = new BootstrapDatagridWrapper(new TDataGrid);
$this->datagrid->style = 'width: 100%';

// cria as colunas
$code = new TDataGridColumn('code',           'Code',      'right', '20%');
$desc = new TDataGridColumn('description', 'Description', 'left',   '40%');
$stock = new TDataGridColumn('stock',        'Stock',     'right', '40%');

// altera o estilo da coluna inteira
$stock->setDataProperty('style','font-weight: bold');

// define o método de transformação
$stock->setTransformer(array($this, 'formatSalary'));

// adiciona as colunas nadatagrid
$this->datagrid->addColumn($code);
$this->datagrid->addColumn($desc);
$this->datagrid->addColumn($stock);

$this->datagrid->createModel(); // cria o modelo dadatagrid
parent::add($this->datagrid); // adiciona adatagrid à página
}

```

O objetivo do método `formatSalary()` é retornar a coluna “`stock`” transformada. Neste exemplo, a transformação que será realizada é a formatação das casas decimais, por meio da função `number_format()`, e também da alteração da cor, conforme o valor (azul para positivos e vermelho para negativos).

O método `formatSalary()` recebe por padrão: o valor da coluna (`$stock`), o objeto Active Record (`$object`), e a linha adicionada nadatagrid (`$row`). Neste exemplo, também é alterado o fundo das linhas com valores negativos (atributo `style`).

```

public function formatSalary($stock, $object, $row)
{
    $number = number_format($stock, 2, ',', '.');
    if ($stock > 0) {
        return "<span style='color:blue'>$number</span>";
    } else {
        $row->style = "background: #FFF9A7";
        return "<span style='color:red'>$number</span>";
    }
}

```

No método `onReload()`, adatagrid é preenchida. Não podemos esquecer também de reescrever o método `show()` da página, para chamar o método `onReload()`, para que adatagrid seja preenchida sempre que a página for exibida.

```

function onReload()
{
    $this->datagrid->clear();

    // adiciona os objetos nadatagrid
    $item = new StdClass;
    $item->code      = '1';
    $item->description = 'Chocolate';
    $item->stock     = 4500;
    $this->datagrid->addItem($item);

    // adiciona demais objetos ...
}

```

```

function show()
{
    $this->onReload();
    parent::show();
}
}

```

Na figura a seguir, podemos ver o resultado da execução deste programa.

Datagrid com formatação		
MARCOS ANTONIO RAFAEL DA FONSECA -		
Code	Description	Stock
1	Chocolate	4.500,00
2	Milk	2.300,00
3	Beer	-500,00
4	Cofee	2.500,00

MARCOS ANTONIO RAFAEL DA FONSECA -

footer

Figura 70 Datagrid com formatação de coluna

4.6.12 Datagrids com links

Neste próximo exemplo, vamos demonstrar como criar umadatagrid com atalhos (links). Neste caso, teremos colunas para e-mail e número de telefone. Sobre a coluna de e-mail, criaremos um atalho para outro programa, que na verdade é uma janela de envio de e-mails (`SingleEmailForm`). Esta janela será aberta por cima dadatagrid, e virá preenchida com o e-mail clicado, o que se dá pela chamada do método `onLoad()` passando o atributo `&email`. Já na outra coluna (telefone), criaremos um atalho para abrir o WhatsApp e já iniciar uma conversação com aquele número.

Obs: O exemplo do link para o WhatsApp poderá deixar de funcionar futuramente caso a empresa mantenedora do serviço resolver realizar alterações nos endereços do mesmo.

Ambos os links são formados por transformações realizadas pelo `setTransformer()`. Neste exemplo, utilizamos uma função anônima, declarada na própria chamada da função `setTransformer()`. Esta função será executada para cada linha de dados dadatagrid e será responsável por modificar o valor recebido da coluna.

Sobre a coluna de e-mail, adicionaremos um ícone na forma de envelope, e em seguida um link (`a href`) que carregará a classe `SingleEmailForm`. Como esta classe é uma janela (`TWindow`), será aberta automaticamente por cima dadatagrid. Utilizamos a tag `generator='adianti'` para que este link não recarregue toda a página ao ser executado.

Já sobre a coluna de telefone, adicionamos um ícone do WhatsApp na frente, e vinculamos ele com o endereço da API do WhatsApp, passando o número de telefone limpo (sem separadores), e acrescido do código do país 55 na frente. Este endereço deve abrir uma conversação em uma nova aba (`target='newwindow'`) do navegador.

app/control/Presentation/Datagrid/DatagridLinkView.class.php

```

<?php
class DatagridLinkView extends TPage
{
    private $datagrid;

    public function __construct()
    {
        parent::__construct();
        $this->datagrid = new BootstrapDatagridWrapper(new TDataGrid);
        $this->datagrid->disableDefaultClick();
        $this->datagrid->width = '100%';

        // cria as colunas
        $code      = new TDataGridColumn('code',     'Code',     'center', '10%');
        $name      = new TDataGridColumn('name',     'Name',     'left',    '30%');
        $email     = new TDataGridColumn('email',    'Email',    'left',    '30%');
        $phone     = new TDataGridColumn('phone',   'Phone',   'left',    '30%');

        // formata a coluna de e-mail, criando um atalho para outra página
        $email->setTransformer( function ($value) {
            if ($value)
            {
                $icon = "<i class='fa fa-envelope-o' aria-hidden='true'></i>";
                return "{$icon} <a generator='adianti' href='index.php?
class=SingleEmailForm&method=onLoad&scroll=0&email=$value'>$value</a>";
            }
            return $value;
        });

        // formata a coluna de telefone, criando um atalho para o WhatsApp
        $phone->setTransformer( function ($value) {
            if ($value)
            {
                $value = str_replace([' ','-','(',')'], ['',',','',''], $value);
                $icon = "<i class='fa fa-whatsapp' aria-hidden='true'></i>";
                return "{$icon} <a target='_newwindow'
href='https://api.whatsapp.com/send?phone=55{$value}&text=Olá'> {$value} </a>";
            }
            return $value;
        });

        // adiciona as colunas
        $this->datagrid->addColumn($code);
        $this->datagrid->addColumn($name);
        $this->datagrid->addColumn($email);
        $this->datagrid->addColumn($phone);

        $this->datagrid->createModel();

        parent::add( TPanelGroup::pack('', $this->datagrid) );
    }

    function onReload()
    {
        // ...
    }

    function show()
    {
        $this->onReload();
        parent::show();
    }
}

```

A figura a seguir demonstra o resultado deste programa.

Code	Name	Email	Phone
MARCOS ANTONIO RAFAEL DA FONSECA -			
1	Aretha Franklin	aretha@email.com	5111111111
2	Eric Clapton	eric@email.com	5122222222
3	B.B. King	king@email.com	5133333333
4	Janis Joplin	janis@email.com	5144444444 MARCOS ANTONIO RAFAEL DA FONSECA -

Figura 71 Datagrid com links

4.6.13 Datagrids com cálculos

Com bastante frequência, datagrids são utilizadas para apresentar valores monetários, como informações financeiras, de compra, de venda, dentre outros. Nestes casos, é preciso formatar os valores com as devidas máscaras contendo a moeda e os corretos separadores decimal e milhar. Além de formatar os valores corretamente, frequentemente é necessário realizar alguma totalização, seja por meio de cálculo entre colunas, ou total geral. Neste exemplo, será demonstrado como aplicar máscara de formatação, cálculo entre colunas, e totalização em uma coluna da datagrid.

O exemplo começa com a criação de uma datagrid com algumas colunas (código, descrição, quantidade, preço, desconto e subtotal). Perceba que a coluna de subtotal contém uma fórmula em seu primeiro parâmetro. Esta fórmula inicia com o caractere de “=”, e pode conter operadores matemáticos ou parêntesis (precedências), e colunas referenciadas entre chaves, como por exemplo `{price}` ou `{amount}`.

Após criarmos as colunas, criamos uma função anônima. Ao atribuirmos a função anônima a uma variável, podemos reaproveitá-la, passando-a como parâmetro, outras vezes. O objetivo da função armazenada em `$format_value` é adicionar o símbolo da moeda, bem como formatar com os corretos separadores decimal e milhar. Essa função será atribuída como função de transformação por meio do método `setTransformer()` para as colunas `$price`, `$discount`, e `$subtotal`.

Após definirmos métodos de transformação para as colunas, utilizamos o método `setTotalFunction()` para definir um método de cálculo de total. Sempre que uma coluna tiver um método de totalização, será acrescentada uma nova linha ao final. Um método de total recebe um vetor com os valores daquela coluna, e produz um resultado. Neste caso, o resultado é simplesmente a soma dos valores por meio da função `array_sum()`, mas podemos facilmente produzir outros totais, como média, etc.

app/control/Presentation/Datagrid/DatagridMathView.class.php

```
<?php
class DatagridMathView extends TPage
{
    private $datagrid;
```

```

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

    $this->datagrid = new BootstrapDatagridWrapper(new TDataGrid);
    $this->datagrid->style = 'width: 100%';

    // cria as colunas da datagrid
    $code      = new TDataGridColumn('code',           'Code',           'right');
    $description = new TDataGridColumn('description', 'Description', 'left');
    $amount     = new TDataGridColumn('amount',         'Amount',        'center');
    $price      = new TDataGridColumn('price',          'Price',          'right');
    $discount   = new TDataGridColumn('discount',       'Discount',       'right');
    $subtotal   = new TDataGridColumn('= {amount} * ({price} - {discount})', 'Subtotal', 'right');

    // adiciona as colunas na datagrid
    $this->datagrid->addColumn($code);
    $this->datagrid->addColumn($description);
    $this->datagrid->addColumn($amount);
    $this->datagrid->addColumn($price);
    $this->datagrid->addColumn($discount);
    $this->datagrid->addColumn($subtotal);

    // define uma função de formatação
    $format_value = function($value) {
        if (!is_numeric($value)) {
            return 'R$ '.number_format($value, 2, ',', '.');
        }
        return $value;
    };

    // atribui a função de formatação às colunas
    $price->setTransformer( $format_value );
    $discount->setTransformer( $format_value );
    $subtotal->setTransformer( $format_value );

    // define uma função de totalização
    $subtotal->setTotalFunction( function($values) {
        return array_sum($values);
    });

    $this->datagrid->createModel();

    // adiciona adatagrid a página
    parent::add($this->datagrid);
}

```

Para carregar alguns dados à datagrid, criamos o método `onReload()`, executado sempre no método `show()`, com alguns dados estáticos para demonstração.

```

function onReload()
{
    $this->datagrid->clear();

    // adiciona um objeto à datagrid
    $item = new StdClass;
    $item->code      = '1';
    $item->description = 'Chocolate';
    $item->amount     = 10;
    $item->price      = 1;
    $item->discount   = 0.05;
    $this->datagrid->addItem($item);
    ...
}

```

```

function show()
{
    $this->onReload();
    parent::show();
}
}

```

Na figura a seguir, podemos ver o resultado da execução deste programa.

Datagrid com cálculos					
MARCOS ANTONIO RAFAEL DA FONSECA -					
Code	Description	Amount	Price	Discount	Subtotal
1	Chocolate	10	R\$ 1,00	R\$ 0,05	R\$ 9,50
2	Milk	20	R\$ 2,00	R\$ 0,50	R\$ 30,00
3	Beer	10	R\$ 4,00	R\$ 1,00	R\$ 30,00
4	Cofee	20	R\$ 2,50	R\$ 0,50	R\$ 40,00
					R\$ 109,50

MARCOS ANTONIO RAFAEL DA FONSECA -					
footer					

Figura 72 Datagrid com cálculos matemáticos

4.6.14 Datagrids com imagem

No exemplo anterior, vimos como formatar valores monetários para a correta exibição em datagrids. Neste exemplo, demonstraremos como exibir imagens. Na maioria das vezes, somente os caminhos das imagens são armazenados no banco de dados. Porém, ao apresentar nadatagrid, gostaríamos de exibir a imagem em si, não o seu caminho. Para tal, podemos aplicar uma função de transformação para gerar uma imagem (objeto **TImage**), a partir da informação do caminho.

Para demonstrarmos tal funcionalidade, vamos criar umadatagrid (**TDataGrid**) e algumas colunas (**code**, **description**, **image**). Sobre a coluna “**image**”, aplicaremos uma transformação. Neste exemplo, a transformação será aplicada por meio de uma função anônima, que é uma função sem nome próprio e que é definida em tempo de execução, podendo ser passada como parâmetro para outros métodos. É importante lembrar que essa função não é executada neste instante, mas sim posteriormente sobre cada uma das linhas dadatagrid, quando estas forem apresentadas. Como ela é aplicada sobre a coluna da imagem, ela simplesmente receberá o caminho (**\$image**), e retornará um objeto do tipo **TImage**, a partir dela.

app/control/Presentation/Datagrid/DatagridImageView.class.php

```

<?php
class DatagridImageView extends TPage
{
    private $datagrid;
    public function __construct()
    {
        parent::__construct();

```

```

// cria adatagrid
$this->datagrid = new BootstrapDatagridWrapper(new TDataGrid);
$this->datagrid->width = '100%';

// adiciona as colunas
$this->datagrid->addColumn( new TDataGridColumn('code','Code','center', '20%' ));
$this->datagrid->addColumn( new TDataGridColumn('description', ... ) );
$column = $this->datagrid->addColumn( new TDataGridColumn('image', 'Image',
'center', '40%' ) );

// define a transformação que rodará sobre a coluna de imagem
$column->setTransformer( function($image) {
    $image = new TImage($image);
    $image->style = 'max-width: 140px';
    return $image;
});

// cria a estrutura dadatagrid em memória
$this->datagrid->createModel();

parent::add($this->datagrid);
}

```

O método `onReload()` será responsável pela carga dos dados. Em exemplos reais, os dados serão carregados a partir do banco de dados. Neste, utilizamos dados estáticos. Veja que a posição “`image`” contém o caminho da imagem.

```

function onReload()
{
    $this->datagrid->clear();

    // adiciona um objeto de dados
    $item = new StdClass;
    $item->code      = '1';
    $item->description = 'Pendrive';
    $item->image     = 'images/pendrive.jpg';
    $this->datagrid->addItem($item);

    ...
}
...
}

```

Na figura a seguir, podemos ver o resultado da execução deste programa.

Datagrid com imagem		
MARCOS ANTONIO RAFAEL DA FONSECA -		
Code	Description	Image
1	Pendrive	

Figura 73 Datagrid com imagem

4.6.15 Datagrids com barra de progresso

No exemplo anterior, vimos como é fácil aplicar uma função de transformação para converter caminhos em imagens. Neste próximo exemplo, veremos como transformar números em barras de progresso. Barras de progresso são muito utilizadas para indicar o percentual de conclusão de tarefas.

Para demonstrar o funcionamento de barras de progresso em datagrids, vamos criar umadatagrid contendo uma coluna numérica de percentual. Nesta coluna, teremos um número inteiro entre 0 e 100, que será apresentado como uma barra de progresso.

Para “gerar” a barra de progresso, vamos definir uma função anônima de transformação sobre a coluna “percent”, por meio do método `setTransformer()`, da mesma maneira que fizemos no exemplo sobre a exibição de imagens. A função de transformação será aplicada somente quando os itens forem apresentados.

A função de transformação recebe como parâmetro o próprio dado adicionado nadatagrid (`$percent`). Então, ela cria um objeto da classe `TProgressBar` e chama seu método `setValue()` para regular o “tamanho” da barra de progresso. O método `setMask()` é utilizado para definir a “máscara” de texto que será exibido sobre a barra de progresso. No `setMask()` podemos utilizar nomes de campos entre chaves. Após, realizamos uma bateria de IF para testar o intervalo numérico que o valor se encontra. Conforme o valor, alteraremos a classe CSS (`success`, `info`, `warning`) da barra de progresso, o que regulará a sua cor.

app/control/Presentation/Datagrid/DatagridProgressView.class.php

```
<?php
class DatagridProgressView extends TPage
{
    private $datagrid;

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

        // cria adatagrid
        $this->datagrid = new BootstrapDatagridWrapper(new TDataGrid);
        $this->datagrid->style = 'width: 100%';

        // adiciona as colunas
        $this->datagrid->addColumn( new TDataGridColumn('code', 'Code', 'center' ) );
        $this->datagrid->addColumn( new TDataGridColumn('task', 'Task', 'left' ) );
        $column = $this->datagrid->addColumn( new TDataGridColumn('percent', ... ) );

        // aplica função de transformação
        $column->setTransformer( function($percent) {
            $bar = new TProgressBar;
            $bar->setMask('Task <b>{value}</b>% complete');
            $bar->setValue($percent);

            if ($percent == 100) {
                $bar->setClass('success');
            }
        } );
    }
}
```

```

        else if ($percent >= 75) {
            $bar->setClass('info');
        }
        else if ($percent >= 50) {
            $bar->setClass('warning');
        }
        else {
            $bar->setClass('danger');
        }
        return $bar;
    });

    $this->datagrid->createModel();
    parent::add($this->datagrid);
}

function onReload()
{
    $this->datagrid->clear();

    // adiciona um objeto de dados
    $item = new StdClass;
    $item->code      = '1';
    $item->task      = 'Install Ubuntu Server';
    $item->percent   = '100';
    $this->datagrid->addItem($item);

    // adiciona um objeto de dados
    $item = new StdClass;
    $item->code      = '2';
    $item->task      = 'Install Apache';
    $item->percent   = '80';
    $this->datagrid->addItem($item);
}

function show()
{
    $this->onReload();
    parent::show();
}
}

```

Na figura a seguir, podemos ver o resultado da execução deste programa.

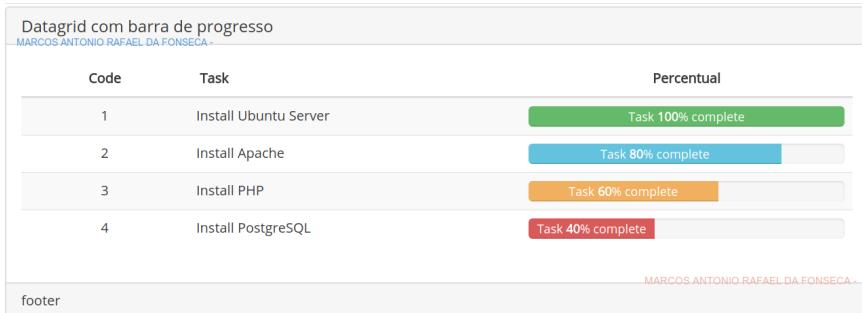


Figura 74 Datagrid com barra de progresso

4.6.16 Exportação de datagrids

Neste exemplo, mostraremos como criar botões que permitem exportar adatagrid em planilha CSV e também em PDF. Este recurso permite exportar a área visível dadatagrid. Para exportação de grandes quantidades de registros, veja o tópico sobre relatórios.

Neste exemplo, criamos umadatagrid convencional, com a exceção de dois botões no topo do painel: um para exportar em PDF e outro para exportar em CSV. Para adicionar botões de ação no painel, utilizamos o método `addHeaderActionLink()`. Este método recebe o label da ação, a ação em si (`TAction`), e o ícone utilizado. No método construtor da ação, além da identificação do método a ser acionado (primeiro parâmetro), no segundo parâmetro podemos indicar parâmetros opcionais. O parâmetro `register_state = false` basicamente indica para o Framework que essa execução não deve se registrada, ou seja, não deve alterar o estado da URL no navegador. A primeira ação (exportação para PDF) executará o método `exportAsPDF()`, já a segunda ação (exportação para CSV) executará o método `exportAsCSV()`.

app/control/Presentation/Datagrid/DatagridExportView.class.php

```
<?php
class DatagridExportView extends TPage
{
    private $datagrid;

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

        $this->datagrid = new BootstrapDatagridWrapper(new TDataGrid);
        $this->datagrid->style = 'width:100%';

        // cria as colunas
        $code      = new TDataGridColumn('code',     'Code',     'center', '10%');
        $name     = new TDataGridColumn('name',     'Name',     'left',   '30%');
        $city     = new TDataGridColumn('city',     'City',     'left',   '30%');
        $state    = new TDataGridColumn('state',    'State',    'left',   '30%');
        // adiciona as colunas
        $this->datagrid->addColumn($code);
        $this->datagrid->addColumn($name);
        $this->datagrid->addColumn($city);
        $this->datagrid->addColumn($state);

        // cria a estrutura
        $this->datagrid->createModel();

        // cria o painel com adatagrid
        $panel = new TPanelGroup(_t('Datagrid export'));
        $panel->add( $this->datagrid );
        $panel->addFooter('footer');

        // adiciona ação para exportar em PDF no topo do painel
        $panel->addHeaderActionLink( 'Save as PDF', new TAction([$this, 'exportAsPDF']),
            ['register_state' => 'false']), 'fa:file-pdf-o red' );

        // adiciona ação para exportar em CSV no topo do painel
        $panel->addHeaderActionLink( 'Save as CSV', new TAction([$this, 'exportAsCSV']),
            ['register_state' => 'false']), 'fa:table blue' );
    }
}
```

```

    parent::add($panel);
}

function onReload()
{
    ...
}

```

O método `exportAsPDF()` gerará um arquivo PDF resultante dadatagrid em tela. Para tal, ele clona o objetodatagrid, e sobre o resultado, utiliza o método `getContents()` para extrair o HTML. Em seguida, usa a biblioteca Dompdf para converter este HTML para PDF, mas antes concatena com o conteúdo do arquivo `styles-print.html`, que contém alguns estilos CSS preparados para ajustar o conteúdo para impressão.

Depois de escrever o PDF em um arquivo, uma janela sob demanda é aberta pelo método `TWindow::create()`. Então, o PDF é embarcado dentro desta janela na forma de um objeto. Este método de conversão deverá funcionar para qualquerdatagrid.

```

public function exportAsPDF($param)
{
    try {
        // extrai o HTML do objeto
        $html = clone $this->datagrid;
        $contents = file_get_contents('app/resources/styles-print.html') .
                    $html->getContents();

        // converte o HTML em PDF
        $dompdf = new \Dompdf\Dompdf();
        $dompdf->loadHtml($contents);
        $dompdf->setPaper('A4', 'portrait');
        $dompdf->render();

        $file = 'app/output/datagrid-export.pdf';

        // escreve em um arquivo o PDF resultante
        file_put_contents($file, $dompdf->output());

        // abre uma janela com o PDF embarcado
        $window = TWindow::create('Export', 0.8, 0.8);
        $object = new TElement('object');
        $object->data = $file;
        $object->type = 'application/pdf';
        $object->style = "width: 100%; height:calc(100% - 10px)";
        $window->add($object);
        $window->show();
    }
    catch (Exception $e) {
        new TMessage('error', $e->getMessage());
    }
}

```

O segundo método `exportAsCSV()` é responsável pela exportação dadatagrid em planilha CSV. Para tal, ele utiliza o método `getOutputData()` dadatagrid. Este método retorna os dados dadatagrid em forma de vetor já processado pelos transformers. Tendo este vetor em mãos, basta percorrê-lo com um `foreach` e utilizar o trio de funções do PHP para escrita em arquivos CSV: `fopen()`, `fputcsv()`, e `fclose()`. Por fim, utilizamos a função `openFile()` para forçar o download do arquivo após sua exportação.

```

public function exportAsCSV($param)
{
    try
    {
        // obtém os dados processados dadatagrid
        $data = $this->datagrid->getOutputData();

        if ($data)
        {
            $file    = 'app/output/datagrid-export.csv';
            $handler = fopen($file, 'w');

            // percorre os dados, escrevendo no CSV
            foreach ($data as $row)
            {
                fputcsv($handler, $row);
            }

            fclose($handler);
            parent::openFile($file);
        }
    }
    catch (Exception $e)
    {
        new TMessage('error', $e->getMessage());
    }
}

function show()
{
    $this->onReload();
    parent::show();
}
}

```

A figura a seguir demonstra a execução deste programa.

Exportação de datagrid			
MARCOS ANTONIO RAFAEL DA FONSECA -			
Code	Name	City	State
1	Aretha Franklin	Memphis	Tennessee (US)
2	Eric Clapton	Ripley	Surrey (UK)
3	B.B. King	Itta Bena	Mississippi (US)
4	Janis Joplin	Port Arthur	Texas (US)

footer

MARCOS ANTONIO RAFAEL DA FONSECA -

Figura 75 Exportação de datagrids

4.6.17 Datagrids com campos de entrada

Neste exemplo, será demonstrado como criar umadatagrid com campos de entrada de dados (**TEntry**). Para tal, tomaremos uma das colunas dadatagrid, que neste caso será o endereço, e a transformaremos em um objeto **TEntry** para digitação. Neste exemplo, a transformação ocorrerá dentro do método **onReload()**, antes do objeto de dados ser adicionado àdatagrid.

No método construtor, adatagrid (**TDataGrid**) é construída com quatro colunas (**code**, **name**, **address**, e **phone**). Logo em seguida criamos um painel (**TPanelGroup**) que será disposto ao redor dadatagrid, o que ocorre pela chamada do método **add()**. Para que possamos utilizar a postagem de dados, é necessário criar um formulário ao redor de todos os campos. Para facilitar, criamos um formulário (**TForm**), ao redor do painel. Assim, usamos o método **add()** para adicionar o painel ao formulário.

Para acionar a postagem dos dados, precisamos ainda de um botão (**\$button**). Este botão, que está vinculado ao método **onSave()**, é posicionado no rodapé do painel, o que se dá pelo método **addFooter()**.

app/control/Presentation/Datagrid/DatagridInputView.class.php

```
<?php
class DatagridInputView extends TPage
{
    private $form;
    private $datagrid;

    public function __construct()
    {
        parent::__construct();
        $this->form = new TForm; // cria um formulário lógico

        // cria adatagrid, define popover, desabilita click default
        $this->datagrid = new BootstrapDatagridWrapper(new TDataGrid);
        $this->datagrid->style = 'width:100%';
        $this->datagrid->enablePopover('hint', 'Type address of <b>{name}</b>');
        $this->datagrid->disableDefaultClick();

        // adiciona um painel ao formulário, e adatagrid ao painel
        $panel = new TPanelGroup(_t('Datagrid with input fields'));
        $panel->add($this->datagrid);
        $this->form->add($panel);

        // adiciona as colunas
        $this->datagrid->addColumn( new TDataGridColumn('code', 'Code', 'right') );
        $this->datagrid->addColumn( new TDataGridColumn('name', 'Name', 'left','30%' ) );
        $this->datagrid->addColumn( new TDataGridColumn('address', 'Address', ... ) );
        $this->datagrid->addColumn( new TDataGridColumn('phone', 'Phone',....) );

        // cria o modelo
        $this->datagrid->createModel();

        // cria o botão de ação e posiciona ele no painel
        $button = TButton::create('action1', [$this, 'onSave'], 'Save', '...');
        $this->form->addField($button);
        $panel->addFooter($button);
        parent::__add($this->form);
    }
}
```

O método **show()** executará o método **onReload()** antes da carga da página.

```
function show()
{
    $this->onReload();
    parent::show();
}
```

O método `onReload()` adiciona objetos àdatagrid. Os objetos são declarados de maneira estática (`stdClass`), com os seus atributos definidos manualmente. Aqui é importante notar que o atributo `address` é preenchido com um objeto (`TEntry`), e não com um valor. Isto fará com que o objeto seja exibido naquela coluna. Para que este objeto seja preenchido com o conteúdo do endereço, utiliza-se o método `setValue()`. No método construtor do `TEntry` identificamos como este campo será enviado.

Além de adicionar os inputs (`TEntry`) a cada linha dadatagrid, é preciso adicioná-los ao formulário, o que se dá pelo método `addField()`, que adiciona mais um campo para ser manipulado logicamente pelo formulário `TForm`. Campos não identificados por este método, não são enviados pelo formulário.

```
function onReload()
{
    $this->datagrid->clear();

    // adiciona um registro àdatagrid
    $item = new stdClass;
    $item->code      = '1';
    $item->name      = 'Aretha Franklin';
    $item->phone     = '1111-1111';

    $item->address   = new TEntry('address1');
    $item->address->setValue('Memphis, Tennessee');
    $item->address->setSize('100%');
    $this->datagrid->addItem($item);

    $this->form->addField($item->address); // importante!

    // adiciona um registro àdatagrid
    $item = new stdClass;
    $item->code      = '2';
    $item->name      = 'Eric Clapton';
    $item->phone     = '2222-2222';

    $item->address   = new TEntry('address2');
    $item->address->setValue('Ripley, Surrey');
    $item->address->setSize('100%');
    $this->datagrid->addItem($item);

    $this->form->addField($item->address); // importante!

    // adiciona outros registros ...
}
```

O método `onSave()` exibe o resultado da digitação do usuário nos campos de entrada de dados. Aqui, caso soubermos o nome do campo, basta acessarmos o atributo do objeto `$data` diretamente. Como muitas vezes os nomes dos campos serão montados dinamicamente (baseados em seu ID), optamos por utilizar o método `getFields()`, que retorna um vetor indexado pelo nome do campo e contendo os campos do formulário. Então, podemos utilizar o método `getValue()` para retornar o conteúdo de cada campo. A mensagem ao usuário (`$message`) é montada dinamicamente, contendo o valor de todos os campos.

```

public function onSave($param)
{
    $data = $this->form->getData();

    // mantém o formulário preenchido
    $this->form->setData($data);

    $message = '';

    // percorre campo a campo do formulário
    foreach ($this->form->getFields() as $name => $field)
    {
        // somente se for um objeto TEntry
        if ($field instanceof TEntry)
        {
            $message .= " $name: " . $field->getValue() . '<br>';
        }
    }

    new TMessage('info', $message); // exibe a mensagem
}
}

```

Na próxima figura, podemos ver o resultado da execução deste programa.

Datagrid com campos de input			
MARCOS ANTONIO RAFAEL DA FONSECA -			
Code	Name	Address	Phone
1	Aretha Franklin	Memphis, Tennessee	1111-1111
2	Eric Clapton	Ripley, Surrey	2222-2222
3	B.B. King	Itta Bena, Mississippi	3333-3333
4	Janis Joplin	Port Arthur, Texas	4444-4444

MARCOS ANTONIO RAFAEL DA FONSECA -

Figura 76 Datagrids com campos de entrada

4.6.18 Datagrids com checkbox

Como vimos até o momento, as datagrids permitem adicionar ações em linhas, e em colunas. As ações em linhas, possibilitam ao usuário executar uma ação por vez (ex: editar, excluir) sobre um registro. O objetivo deste exemplo é demonstrar como podem ser selecionados vários registros em umadatagrid ao mesmo tempo, por meio de checkboxes. Além de podermos marcar vários registros, teremos também um botão que permitirá enviar os dados dos registros selecionados por um formulário.

Para iniciar, criaremos um formulário e umadatagrid (**TDataGrid**), e logo em seguida, executaremos o método **disableDefaultClick()**. Este método desliga a opção de execução automática da primeira ação dadatagrid, quando o usuário clica sobre a linha, o que é indispensável quando utilizamos checkboxes.

No método construtor, adatagrid (**TDataGrid**) é construída com cinco colunas (**check**, **code**, **name**, **address**, e **phone**). A coluna Check não apresentará um dado, mas um botão de check que será criado posteriormente no método **onReload()**.

Um painel (**TPanelGroup**) será disposto ao redor dadatagrid. Para que possamos utilizar a postagem de dados, é necessário criar um formulário ao redor de todos os campos. Para facilitar, criamos um formulário (**TForm**), ao redor do painel. Assim, usamos o método **add()** para adicionar o painel ao formulário.

Para acionar a postagem dos dados, precisamos ainda de um botão (**\$button**). Este botão, que está vinculado ao método **onSave()**, é posicionado no rodapé do painel, o que se dá pelo método **addFooter()**.

app/control/Presentation/Datagrid/DatagridCheckView.class.php

```
<?php
class DatagridCheckView extends TPage
{
    private $form, $datagrid;
    public function __construct()
    {
        parent::__construct();
        $this->form = new TForm; // cria um formulário

        // cria umadatagrid
        $this->datagrid = new BootstrapDatagridWrapper(new TDataGrid);
        $this->datagrid->style = 'width: 100%';
        $this->datagrid->disableDefaultClick(); // importante!

        // adiciona adatagrid ao painel, e o painel ao form
        $panel = new TPanelGroup(_t('Datagrids with Checkbutton'));
        $panel->add($this->datagrid)->style = 'overflow-x:auto';
        $this->form->add($panel);

        // adiciona as colunas dadatagrid
        $this->datagrid->addColumn( new TDataGridColumn('check', 'Check', 'right',...));
        $this->datagrid->addColumn( new TDataGridColumn('code', 'Code', 'right',...));
        $this->datagrid->addColumn( new TDataGridColumn('name', 'Name', 'left',...));
        $this->datagrid->addColumn( new TDataGridColumn('address', 'Address', ...));
        $this->datagrid->addColumn( new TDataGridColumn('phone', 'Phone', ...));

        $this->datagrid->createModel(); // cria o modelo dadatagrid

        // cria um botão de ação
        $button = TButton::create('action1', [$this, 'onSave'], 'Save', '...');
        $this->form->addField($button);
        $panel->addFooter($button);

        parent::__add($this->form);
    }
}
```

O método **onReload()** carregará os dados nadatagrid. E é neste método que está parte do segredo do funcionamento deste exemplo. No método construtor, criamos as colunas dadatagrid, e uma delas chama-se “**check**”. Pois bem, no método **onReload()** no lugar de atribuir um valor escalar (inteiro, string) ao atributo “**check**”, atribuiremos um objeto da classe **TCheckButton**. Isto fará com que este objeto seja exibido dentro da coluna. Cada objeto terá um nome diferente (**check1**, **check2**, etc.).

Como este objeto deverá fazer parte do formulário que contém a datagrid, precisamos executar o método `addField()` novamente, para que o formulário reconheça este objeto e passe a manipulá-lo em postagens.

```
function onReload()
{
    $this->datagrid->clear();

    // adiciona um registro nadatagrid
    $item = new StdClass;
    $item->check  = new TCheckButton('check1');
    $item->check->setIndexValue('on');
    $item->code   = '1';
    $item->name   = 'Aretha Franklin';
    $item->address = 'Memphis, Tennessee';
    $item->phone  = '1111-1111';
    $this->datagrid->addItem($item);
    $this->form->addField($item->check); // importante!

    // adiciona um registro nadatagrid
    $item = new StdClass;
    $item->check  = new TCheckButton('check2');
    $item->check->setIndexValue('on');
    $item->code   = '2';
    $item->name   = 'Eric Clapton';
    $item->address = 'Ripley, Surrey';
    $item->phone  = '2222-2222';
    $this->datagrid->addItem($item);
    $this->form->addField($item->check); // importante!

    // adiciona demais objetos
}
```

O objetivo do método `onSave()` é somente exibir quais botões de checagem foram selecionados. Para tal, ele obtém os dados do formulário e monta uma string contendo cada botão de checagem (check1, check2, check3, etc...). Após, exibe um diálogo para mostrar os resultados para o usuário.

```
public function onSave($param)
{
    $data = $this->form->getData(); // obtém os dados do form
    $this->form->setData($data); // mantém o formulário preenchido

    // cria uma string com as opções selecionadas
    $message = 'Check 1 : ' . $data->check1 . '<br>';
    $message.= 'Check 2 : ' . $data->check2 . '<br>';
    $message.= 'Check 3 : ' . $data->check3 . '<br>';
    $message.= 'Check 4 : ' . $data->check4 . '<br>';

    new TMessage('info', $message);
}
```

O método `show()` deve ser reescrito para executar o método `onReload()`.

```
function show()
{
    $this->onReload();
    parent::show();
}
```

Na figura a seguir, podemos conferir o resultado da execução deste programa.

Datagrid com checkbox				
MARCOS ANTONIO RAFAEL DA FONSECA -				
Check	Code	Name	Address	Phone
<input type="checkbox"/>	1	Aretha Franklin	Memphis, Tennessee	1111-1111
<input type="checkbox"/>	2	Eric Clapton	Ripley, Surrey	2222-2222
<input type="checkbox"/>	3	B.B. King	Itta Bena, Mississippi	3333-3333
<input type="checkbox"/>	4	Janis Joplin	Port Arthur, Texas	4444-4444

MARCOS ANTONIO RAFAEL DA FONSECA -

Figura 77 Datagrids com checkbox

4.7 Transições de páginas

Na maioria dos sistemas, é necessário construir interfaces que requerem que os dados do usuário sejam informados aos poucos, por meio de telas de passo a passo, ou wizards. Estas telas são comuns na criação de contas de usuários, ou de pesquisas de opinião, dentre outros. Nesta seção, veremos como construir telas passo a passo.

4.7.1 Passo a passo entre formulários diferentes

O objetivo deste exemplo é demonstrar como criar uma navegação, com botões de avançar e retroceder, entre diferentes formulários. O primeiro formulário tem como objetivo coletar os dados de login de um usuário (e-mail e senha). Esta tela direcionará o usuário para um próximo formulário que perguntará os dados pessoais do usuário (Nome, Sobrenome, Telefone). Todo o processo tem como finalidade simular a criação de uma conta de usuário.

No método construtor do primeiro formulário criamos a tela, declarando os campos (e-mail, senha e confirmação) dentro de um formulário (`BootstrapFormBuilder`). Este formulário possui apenas um botão de avançar, vinculado ao método `onNextForm()`. Enquanto que os campos são adicionados pelo `addFields()`, o botão de ação é criado pelo `addAction()`. Algumas validações também são adicionadas aos campos.

app/control/Presentation/Pages/MultiStepMultiFormView.class.php

```
<?php
class MultiStepMultiFormView extends TPage
{
    protected $form; // form

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

        // cria o formulário
        $this->form = new BootstrapFormBuilder('form_account');
        $this->form->setFormTitle('Create account');
        $this->form->setColumnClasses(2, ['col-sm-3', 'col-sm-9']);
    }
}
```

```

// cria os campos do formulário
$email      = new TEntry('email');
$password   = new TPASSWORD('password');
$confirm    = new TPASSWORD('confirm');

$this->form->addFields(['Email' , [$email] ]);
$this->form->addFields(['Password' , [$password] ]);
$this->form->addFields(['Confirm password' , [$confirm] ]);
// adiciona algumas validações
$email->addValidation('Email', new TRequiredValidator);
$email->addValidation('Email', new TEmailValidator);
$password->addValidation('Password', new TRequiredValidator);
$confirm->addValidation('Confirm password', new TRequiredValidator);

// adiciona a ação
$this->form->addAction('Next', new TAction(array($this, 'onNextForm')),
                           'fa:chevron-circle-right green');
parent::add($this->form);
}

```

O método `onNextForm()` valida o formulário, compara as senhas digitadas, armazena os dados digitados em uma variável de sessão (`form_step1_data`), por meio do método `TSession::setValue()`, e direciona o usuário para outro formulário (`MultiStepMultiForm2View`), por meio do método `AdiantiCoreApplication::loadPage()`, que executará o método `onLoadFromForm1()` do próximo formulário. O vetor `$data` é passado como parâmetro, e estará disponível do outro lado por meio da variável `$param`.

```

public function onNextForm()
{
    try {
        $this->form->validate();
        $data = $this->form->getData();

        if ($data->password !== $data->confirm)
        {
            throw new Exception('Passwords do not match');
        }
        // armazena dados em uma variável de sessão
        TSession::setValue('form_step1_data', $data);

        // Carrega outra página
        AdiantiCoreApplication::loadPage('MultiStepMultiForm2View',
                                         'onLoadFromForm1', (array) $data);
    }
    catch (Exception $e) {
        new TMessage('error', $e->getMessage());
    }
}

```

Já o método `onLoadFromSession()` é executado somente a partir do segundo formulário, quando o usuário desejar voltar para este primeiro formulário. Quando esta operação ocorrer, os dados deverão ser mantidos preenchidos, o que é realizado por meio da leitura da sessão (`TSession::getValue()`).

```

public function onLoadFromSession()
{
    $data = TSession::getValue('form_step1_data');
    $this->form->setData($data);
}

```

A figura a seguir exibe o primeiro formulário da sequência.

The screenshot shows a web form titled "Create account". It contains three input fields: "Email", "Password", and "Confirm password". Below the fields is a "Next" button. A watermark in the background reads "MARCOS ANTONIO RAFAEL DA FONSECA -".

Figura 78 Primeiro formulário do passo a passo

Na segunda página, é construída uma tela contendo um formulário (**BootstrapFormBuilder**) com campos para Nome, sobrenome, além do telefone. Este formulário terá dois botões de ação: um para confirmar o cadastro, que estará vinculado ao método `onConfirm()`, e outro para regressar à tela anterior, vinculado ao método `onBackForm()`. Este formulário será carregado a partir da classe anterior por meio do método `onLoadFromForm1()`, que passará alguns dados iniciais via parâmetro.

app/control/Presentation/PageTransitions/MultiStepMultiForm2View.class.php

```
<?php
class MultiStepMultiForm2View extends TPage
{
    protected $form; // form
    function __construct()
    {
        parent::__construct();

        // cria o formulário
        $this->form = new BootstrapFormBuilder('form_account');
        $this->form->setFormTitle('Personal details');
        $this->form->setColumnClasses(2, ['col-sm-3', 'col-sm-9'] );

        // cria os campos
        $email      = new TEntry('email');
        $first_name = new TEntry('first_name');
        $last_name  = new TEntry('last_name');
        $phone      = new TEntry('phone');
        $email->setEditable(FALSE);

        // adiciona os campos ao formulário
        $this->form->addFields(['Email'], [$email] );
        $this->form->addFields(['First name'], [$first_name] );
        $this->form->addFields(['Last name'], [$last_name] );
        $this->form->addFields(['Phone'], [$phone] );

        // adiciona validações
        $first_name->addValidation('First name', new TRequiredValidator);
        $last_name->addValidation('Last name', new TRequiredValidator);
        $phone->addValidation('Phone', new TRequiredValidator);
    }
}
```

```

    // adiciona as ações de voltar e confirmar
    $this->form->addAction('Back', new TAction(array($this, 'onBackForm')),
        'fa:chevron-circle-left orange');
    $this->form->addAction('Confirm', new TAction(array($this, 'onConfirm')),
        'fa:check-circle-o green');

    parent::add($this->form); // adiciona o formulário à página
}

```

O método `onLoadFromForm1()` é executado a partir do primeiro formulário, onde existe um botão vinculado a ele. Este método é responsável por preencher o e-mail do usuário no segundo formulário, com base no e-mail preenchido no primeiro formulário, por meio do método `TForm::setData()`. Este método recebe os dados passados como parâmetro (`$data`), que foram preenchidos no primeiro formulário.

```

public function onLoadFromForm1($data)
{
    $obj = new StdClass;
    $obj->email = $data['email'];
    $this->form->setData($obj);
}

```

O método `onBackForm()` é responsável por regressar ao formulário anterior, o que é feito por meio do método `AdiantiCoreApplication::loadPage()`, identificando a classe e o método a serem executados. O método `MultiStepMultiFormView::onLoadFromSession()` faz parte do primeiro formulário, e naquela classe é responsável por preencher o formulário com os dados da sessão.

```

public function onBackForm()
{
    // volta para o formulário anterior
    AdantiCoreApplication::loadPage('MultiStepMultiFormView', 'onLoadFromSession');
}

```

Já o método `onConfirm()` é responsável por obter os dados digitados no segundo formulário, agregar a senha digitada no primeiro formulário, que é lida da sessão pelo método `TSession::getValue()`, e exibir um diálogo de informação (`TMessage`), contendo os dados do usuário, que foram coletados pelos dois formulários.

```

public function onConfirm()
{
    try
    {
        $this->form->validate();
        $data = $this->form->getData();
        $this->form->setData($data);

        $form1_data = TSession::getValue('form_step1_data');
        $data->password = $form1_data->password; // agrupa a senha
        new TMessage('info', str_replace(',', ', <br>', json_encode($data)));
    }
    catch (Exception $e)
    {
        new TMessage('error', $e->getMessage());
    }
}

```

Na próxima figura, podemos ver o segundo formulário da sequência.

Personal details
MARCOS ANTONIO RAFAEL DA FONSECA -

Email	pablo@dalloglio.net
First name	Pablo
Last name	Dall Oggio
Phone	123123123123

MARCOS ANTONIO RAFAEL DA FONSECA -

Figura 79 Segundo formulário do passo a passo

4.7.2 Wizard passo a passo

O objetivo deste exemplo é demonstrar a criação de um Wizard (assistente) com passo a passo entre diferentes tipos de tela. Neste assistente, teremos 4 telas que serão: apresentação → escolha do curso → digitação dos dados → confirmação. A primeira tela (apresentação) e a última (confirmação) serão telas de exibição de HTML, sendo a primeira de boas vindas, e a última para apresentar os dados selecionados. A segunda tela (escolha do curso) será umadatagrid de seleção de curso, e a terceira (digitação dos dados) será um formulário para entrada de dados de identificação.

A primeira tela da sequência será somente uma caixa de boas vindas com um texto padrão qualquer e um botão de continuar. Para tal, utilizaremos a classe `THtmlRenderer` para exibir um fragmento de HTML e tela (`welcome.html`). Já a classe `TPageStep` é utilizada para criar uma trilha de navegação superior (passo a passo). O método `addItem()` adiciona um item na trilha, e o método `select()` indica qual item da trilha é o atual.

[app/control/Presentation/Pages/MultiStepRegistration1View.class.php](#)

```
<?php

class MultiStepRegistration1View extends TPage
{
    public function __construct()
    {
        parent::__construct();
        try {
            // cria o renderizador de HTML
            $this->html = new THtmlRenderer('app/resources/welcome.html');
            $this->html->enableSection('main');

            // cria uma trilha de navegação (passo a passo)
            $pagestep = new TPageStep;
            $pagestep->addItem('Welcome');
            $pagestep->addItem('Selection');
            $pagestep->addItem('Complete information');
            $pagestep->addItem('Confirmation');
            $pagestep->select('Welcome');
```

```

    // empacota tudo em uma caixa vertical
    $vbox = new TBox;
    $vbox->style = 'width: 100%';
    $vbox->>add( $pageStep );
    $vbox->>add($this->html);

    // adiciona o conteúdo à página
    parent::add($vbox);
}
catch (Exception $e)
{
    new TMessage('error', $e->getMessage());
}
}

public function loadPage()
{
}
}
}

```

A figura a seguir demonstra a primeira tela do assistente.

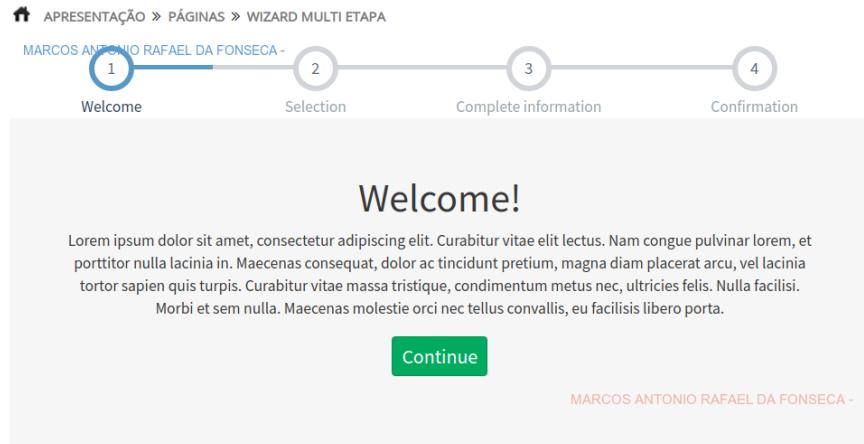


Figura 80 Tela de boas vindas

A segunda página do assistente terá uma datagrid na qual o usuário terá acesso a uma lista de cursos, e terá somente de clicar em um deles para progredir para a próxima etapa. A datagrid possui uma só ação (`onSelect`), que passa como parâmetro o código (`code`) e a descrição (`description`) do curso selecionado.

Esta tela também terá uma trilha (`TPageStep`), mas desta vez, o item selecionado será o segundo (Selection). Esta tela também terá um botão de retorno à página anterior (`TActionLink`), que executa um método por meio de uma requisição GET, levando o usuário à página anterior por meio do método `MultiStepRegistration1View::loadPage()`. O botão de retorno é posicionado logo abaixo da datagrid, no rodapé do painel que envolve ela, por meio do método `TPanelGroup::pack()`.

app/control/Presentation/Pages/MultiStepRegistration2View.class.php

```
<?php

class MultiStepRegistration2View extends TPage
{
    private $datagrid;

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

        // cria a datagrid
        $this->datagrid = new BootstrapDatagridWrapper(new TDataGrid);

        // adiciona as colunas
        $this->datagrid->addColumn( new TDataGridColumn('code','Code','center',...) );
        $this->datagrid->addColumn( new TDataGridColumn('description', 'Description') );

        // cria a ação dadatagrid
        $action1 = new TDataGridAction([$this, 'onSelect'],
            ['code'=>'{code}', 'description' => '{description}']);
        $this->datagrid->addAction($action1, 'Select', 'fa:check-circle-o fa-fw');

        $this->datagrid->createModel();

        // cria o passo a passo
        $pagestep = new TPageStep;
        $pagestep->addItem('Welcome');
        $pagestep->addItem('Selection');
        $pagestep->addItem('Complete information');
        $pagestep->addItem('Confirmation');
        $pagestep->select('Selection');

        // cria o botão de retorno
        $back_action = new TAction(array('MultiStepRegistration1View', 'loadPage'));
        $back = new TActionLink('Back', $back_action, 'black', null, null, 'fa:arrow-circle-o-left red');
        $back->addStyleClass('btn btn-default btn-sm');

        $vbox = new TVBox;
        $vbox->style = 'width: 100%';
        $vbox->add( $pagestep );
        $vbox->add( TPanelGroup::pack('', $this->datagrid, $back) );

        parent::add($vbox);
    }
}
```

O método `onReload()` será executado antes que a página seja exibida (`show`), e adicionará alguns registros à datagrid. Neste exemplo, os registros são pré-definidos.

```
function onReload()
{
    $this->datagrid->clear();

    // adiciona um registro àdatagrid
    $item = new StdClass;
    $item->code      = '1';
    $item->description = 'Intro to Computer Science';
    $this->datagrid->addItem($item);

    // ...
}
```

O método `onSelect()`, recebe os parâmetros da seleção do curso (`code`, `description`), armazena eles em uma variável de sessão (`registration_course`), e carrega a próxima tela, por meio do método `loadPage()`.

```
function onSelect($param)
{
    TSession::setValue('registration_course', ['course_id' => $param['code'],
                                                'course_description' => $param['description']]);
    AdiantiCoreApplication::loadPage('MultiStepRegistration3View');
}
function show()
{
    $this->onReload();
    parent::show();
}
}
```

A figura a seguir demonstra a segunda tela do assistente.

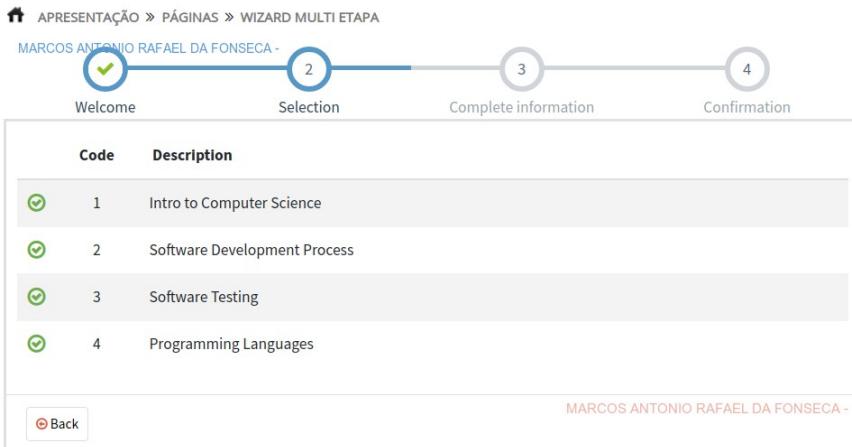


Figura 81 Tela de seleção de curso

A terceira tela do assistente, terá um formulário para que o usuário possa entrar com alguns dados de identificação, como e-mail, nome, sobrenome, e telefone. Em seu método construtor, criamos um formulário (`BootstrapFormBuilder`), e alguns campos para entrada de dados (`TEntry`). Em seguida adicionamos os campos ao formulário (`addFields`), e algumas validações (`addValidation`). Este formulário terá duas ações: voltar, que executará o método `onBackForm()`, e confirmar, que executará o método `onConfirm()`. Por fim, adicionamos uma trilha (`TBreadcrumb`) para posicionar o usuário na sequência de telas do passo a passo.

app/control/Presentation/Pages/MultiStepRegistration3View.class.php

```
<?php
class MultiStepRegistration3View extends TPage
{
    protected $form; // form
```

```

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

    // cria o formulário
    $this->form = new BootstrapFormBuilder('form_account');
    $this->form->setFormTitle('Personal details');

    // cria os campos do formulário
    $email      = new TEntry('email');
    $first_name = new TEntry('first_name');
    $last_name  = new TEntry('last_name');
    $phone      = new TEntry('phone');

    // adiciona os campos ao formulário
    $this->form->addFields(['Email'], [$email] );
    $this->form->addFields(['First name'], [$first_name] );
    $this->form->addFields(['Last name'], [$last_name] );
    $this->form->addFields(['Phone'], [$phone] );

    // adiciona algumas validações
    $email->addValidation('Email', new TEmailValidator);
    $first_name->addValidation('First name', new TRequiredValidator);
    $last_name->addValidation('Last name', new TRequiredValidator);
    $phone->addValidation('Phone', new TRequiredValidator);

    // adiciona as ações do formulário
    $this->form->addAction('Back', new TAction(array($this, 'onBackForm')),
                           'fa:arrow-circle-o-left red');
    $this->form->addAction('Confirm', new TAction(array($this, 'onConfirm')),
                           'fa:check-circle-o green');

    // cria a trilha
    $pagestep = new TPageStep;
    $pagestep->addItem('Welcome');
    $pagestep->addItem('Selection');
    $pagestep->addItem('Complete information');
    $pagestep->addItem('Confirmation');
    $pagestep->select('Complete information');

    // empacota os elementos em uma caixa vertical
    $vbox = new TVBox;
    $vbox->style = 'width: 100%';
    $vbox->add( $pagestep );
    $vbox->add( $this->form );
    parent::add($vbox);
}

```

O método `onConfirm()` armazenará os dados preenchidos pelo usuário no formulário em uma variável de sessão (`registration_data`), e direcionará o usuário para a última etapa do assistente (`MultiStepRegistration4View`).

```

public function onConfirm()
{
    try {
        $this->form->validate();
        $data = $this->form->getData();
        TSession::setValue('registration_data', (array) $data);
        AdiantiCoreApplication::loadPage('MultiStepRegistration4View');
    }
    catch (Exception $e) {
        new TMessage('error', $e->getMessage());
    }
}

```

O método `onBackForm()` permitirá que o usuário seja redirecionado para a página anterior, por meio do método `loadPage()`.

```
public function onBackForm()
{
    // carrega a página do passo anterior
    AdantiCoreApplication::loadPage('MultiStepRegistration2View');
}
```

A figura a seguir demonstra a terceira tela do assistente.

Figura 82 Tela de digitação de dados pessoais

A última tela terá apenas de exibir uma confirmação com os dados das etapas anteriores, que são: e-mail, nome, sobrenome, telefone, e curso selecionados.

A classe `THtmlRenderer` será utilizada para exibir um fragmento de HTML (`confirmation.html`). Neste fragmento, existirão algumas marcações como: `{$email}`, `{$first_name}`, `{$last_name}`, `{$phone}`, `{$course_id}`, e `{$course_description}`, que serão substituídas em tempo de execução pelos dados lidos das variáveis de sessão.

As variáveis de sessão `registration_course` e `registration_data`, contém as informações do curso selecionado, e dos dados de identificação do usuário. Este conteúdo é unificado na variável `$confirmation_data`, por meio da função `array_merge()`.

O conteúdo das variáveis é passado para o HTML por meio do segundo parâmetro do método `enableSection()`, que recebe um vetor de conteúdos para preencher as variáveis do HTML.

Obs: Nas próximas sessões abordaremos templates de maneira mais detalhada.

```
app/control/Presentation/Pages/MultiStepRegistration4View.class.php
<?php

class MultiStepRegistration4View extends TPage
{
    public function __construct()
    {
        parent::__construct();
        try {
            TPage::include_css('app/resources/confirmation.css');

            // cria o renderizador de HTML
            $this->html = new THtmlRenderer('app/resources/confirmation.html');

            // cria um vetor único com os dados de duas variáveis de sessão
            $confirmation_data = array_merge(TSession::getValue('registration_course'),
                TSession::getValue('registration_data'));

            // habilita o template, passando os dados
            $this->html->enableSection('main', $confirmation_data);

            $pagestep = new TPageStep;
            $pagestep->addItem('Welcome');
            $pagestep->addItem('Selection');
            $pagestep->addItem('Complete information');
            $pagestep->addItem('Confirmation');
            $pagestep->select('Confirmation');

            $vbox = new TVBox;
            $vbox->style = 'width: 100%';
            $vbox->add( $pagestep );
            $vbox->add($this->html);
            parent::add($vbox);
        }
        catch (Exception $e) {
            new TMessage('error', $e->getMessage());
        }
    }
}
```

A figura a seguir demonstra a quarta tela do assistente.

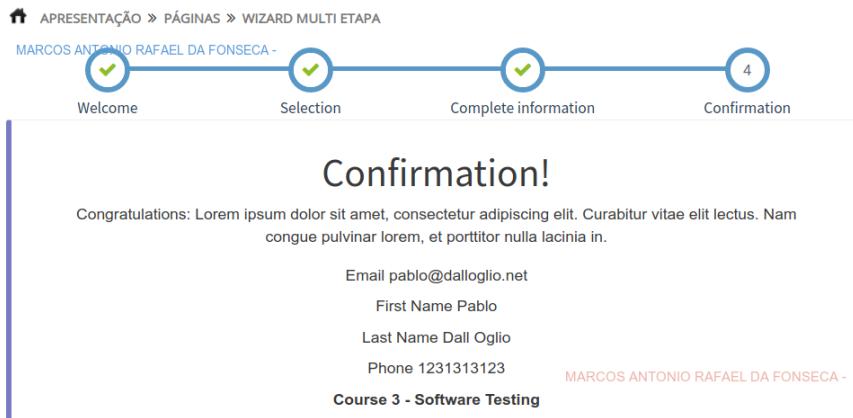


Figura 83 Tela de confirmação do assistente.

4.8 Utilitários

Nesta seção, serão abordados componentes utilitários com a visão de árvore (**TTreeView**), o calendário (**TFullCalendar**), passo a passo (**TPageStep**), Visão de ícones (**TIconView**), Timeline (**TTimeline**), visão de cards (**TCardView**), e o kanban (**TKanban**).

4.8.1 Passo a passo

O primeiro utilitário que vamos abordar é o passo a passo (**TPageStep**). Este componente, é utilizado normalmente na parte superior de uma página que integra um wizard (assistente) passo a passo. Normalmente um passo a passo é formado por várias páginas, sendo que no topo de cada uma, repetimos um **TPageStep**, com os mesmos itens, porém selecionando um item diferente de cada vez. O método **select()**, é utilizado para definir o item atual. Este método deixa o item selecionado com um contorno azul. Os itens seguintes (ainda não executados) ficam em cinza, e os itens anteriores (já executados) ficam com um ok na cor verde para sinalizar que essa etapa já foi cumprida.

O método **addItem()** recebe opcionalmente um segundo parâmetro que representa uma ação (**TAction**). Se preenchido, ele define uma ação a ser executada quando o usuário clicar no label (texto) da etapa.

app/control/Presentation/Widgets/StepView.class.php

```
<?php
class StepView extends TPage
{
    private $step;

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

        // cria o passo a passo
        $this->step = new TPageStep;
        $this->step->addItem('Step 1',
            new TAction(['CustomerDataGridView', 'onReload']));
        $this->step->addItem('Step 2');
        $this->step->addItem('Step 3');

        // seleciona a etapa corrente
        $this->step->select('Step 2');

        parent::add($this->step);
    }
}
```

A figura a seguir demonstra a execução deste programa.



Figura 84 Passo a passo

4.8.2 Visão de ícones

O próximo exemplo, demonstrará como criar uma visão de ícones. Uma visão de ícones é comumente utilizada em apresentação de sistema de arquivos, mas podemos utilizar também na exibição de registros do banco de dados, embora não tão comum.

O exemplo a seguir, será demonstrado de maneira avulsa, sem estar vinculada à sistema de arquivos ou banco de dados. Dessa forma, ao compreender a natureza do componente, você poderá adaptar seu uso sob diferentes circunstâncias.

Para utilizar a visão de ícones, precisamos inicialmente instanciar a classe `TIconView`. Seu funcionamento se assemelha muito ao componente de datagrids, pois definimos um conjunto de atributos para depois adicionar itens ao componente. Como vamos adicionar os dados posteriormente (na forma de objetos), neste primeiro momento, precisamos definir quais os atributos destes objetos serão utilizados. Dessa forma, o método `setIconAttribute()` define qual o atributo dos dados representará o ícone, `setLabelAttribute()` define qual atributo dos dados representará o nome do ícone, e `setInfoAttributes()` define quais atributos serão passados como parâmetro em ações.

Quando clicamos com o botão direito sobre um ícone, um menu de contexto será aberto. Para definir as opções deste menu de contexto, podemos usar o método `addContextMenuOption()`, que recebe o nome da ação a ação em si (objeto `TAction`), e um ícone para representá-la. Existe ainda um quarto parâmetro opcional, que define uma condição (função) que se retornar false, não disponibiliza a opção para o usuário. Neste exemplo, criamos um menu com várias opções, mas todas ligadas ao `onAction()`.

[app/control/Presentation/Widgets/IconView.class.php](#)

```
<?php
class IconView extends TPage
{
    private $iconview;

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

        // cria a visão de ícone
        $this->iconview = new TIconView;
        $this->iconview->setIconAttribute('icon');
        $this->iconview->setLabelAttribute('name');
        $this->iconview->setInfoAttributes(['name', 'path']);

        // habilita popover sobre os ícones
        $this->iconview->enablePopover('', '<b>Name</b>: {name} <br> <b>Path</b>: {path}', 'top');

        // função que define uma condição para exibir ou não a ação
        $display_condition = function($object) {
            return $object->type == 'file';
        };
    }
}
```

```

// adiciona menu de contexto
$this->iconview->addContextMenuOption('Options');
$this->iconview->addContextMenuOption('');

// adiciona uma ação ao menu de contexto
$this->iconview->addContextMenuOption('Acao 1',
    new TAction([$this, 'onAction']), 'fa:folder-o blue');
$this->iconview->addContextMenuOption('Acao 2',
    new TAction([$this, 'onAction']), 'fa:check-circle-o green');
$this->iconview->addContextMenuOption('Acao 3',
    new TAction([$this, 'onAction']), 'fa:trash red', $display_condition);

parent::add($this->iconview);
}

```

Neste exemplo, quando o usuário acionar qualquer uma das opções do menu de contexto do ícone, o método `onAction()` será executado. Neste exemplo, preferimos ligar todas as opções ao mesmo método somente para poupar espaço.

```

public static function onAction($param)
{
    new TMessage('info', '<b>Path: </b>' . $param['path'] .
        '<br> <b> Name: </b>' . $param['name']);
}

```

Objetos de dados precisam ser carregados na visão de ícones. Cada objeto será exibido como um ícone. Isto é feito pelo método `onReload()`, executado sempre dentro do `show()`. Neste método, adicionamos uma série de objetos de dados aos ícones. É importante lembrar que o atributo `icon`, foi definido como ícone, portanto ali utilizamos um ícone da Font Awesome (`fa:folder-o blue fa-4x`). Já o atributo `name` foi definido como descriptivo, e `name` e `path` como parâmetros para ações.

```

function onReload()
{
    // add an regular object to the datagrid
    $item = new StdClass;
    $item->type      = 'folder';
    $item->path      = '/folder-a';
    $item->name      = 'Folder A';
    $item->icon      = 'fa:folder-o blue fa-4x';
    $this->iconview->addItem($item);

    // add an regular object to the datagrid
    $item = new StdClass;
    $item->type      = 'file';
    $item->path      = '/file-a';
    $item->name      = 'File A';
    $item->icon      = 'fa:file-o orange fa-4x';
    $this->iconview->addItem($item);

    ...
}

function show()
{
    $this->onReload();
    parent::show();
}
}

```

A figura a seguir demonstra o resultado da execução deste programa.

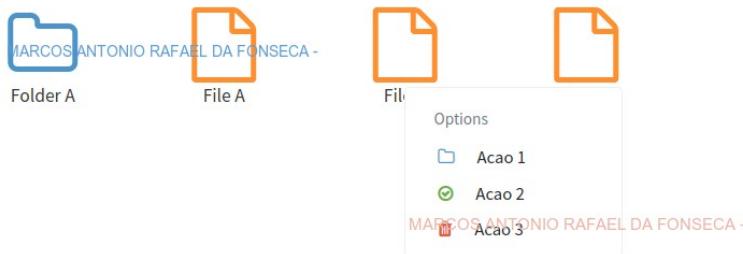


Figura 85 Visão de ícones

4.8.3 Linha do tempo

O próximo exemplo que vamos demonstrar é a linha do tempo (`TTimeline`). Este objeto permite criar visões ordenadas cronologicamente. É um bom componente para ser utilizado para demonstração de eventos que dependam da ordem de tempo para serem exibidos em tela. Alguns exemplos são: conversações entre pessoas, registros de atividades em sistemas, registros de logs, dentre outros.

Para criar uma timeline, inicialmente instanciamos o componente `TTimeline`. Para adicionar eventos na linha do tempo, basta utilizarmos o método `addItem()`. Este método recebe o id do evento, seu título, um conteúdo, data/hora de sua ocorrência, um ícone, seu lado na timeline, e um objeto de dados. Este objeto de dados é usado para representar aquele evento específico, e seus atributos podem ser utilizados como parâmetros em ações. Dentro do título, ou do conteúdo do evento, podemos utilizar atributos como em `{name}`. Estes atributos são alimentados a partir do objeto de dados informado no último parâmetro do `addItem()`.

Obs: A data/hora do evento sempre deve ser informada no formato `yyyy-mm-dd`.

O método `setUseBothSides()` é utilizado para habilitar os dois lados da timeline. Assim, eventos são exibidos tanto à esquerda quanto à direita. O método `setTimeDisplayMask()` define a máscara de exibição para o tempo, e o método `setFinalIcon()` define o ícone final da timeline.

O método `addAction()` é utilizado para adicionar um botão de ação ao item da timeline. Ele recebe uma ação (`TAction`), um rótulo, um ícone, e opcionalmente uma condição de exibição. O último parâmetro (condição) trata-se de uma função que determina quando aquela ação será exibida. A condição de exibição pode se basear por exemplo, em um atributo do objeto. Neste exemplo simples, estamos observando apenas o id [2,3], mas na prática podemos observar atributos de status, e até mesmo variáveis de sessão (Ex. Usuário logado), para verificar se o usuário terá acesso à ação ou não.

app/control/Presentation/Widgets/TimelineView.class.php

```
<?php
class TimelineView extends TPage
{
    public function __construct()
    {
        parent::__construct();

        $timeline = new TTimeline;
        $obj1 = (object) [ 'name' => 'AAA' ];
        $obj2 = (object) [ 'name' => 'BBB' ];
        $obj3 = (object) [ 'name' => 'CCC' ];

        // adiciona itens à timeline
        $timeline->addItem(1, 'Event {id}', 'This is the event id: <b>{id}</b>
name: <b>{name}</b>', '2017-12-11 12:01:00', 'fa:arrow-left bg-green', 'left',
$obj1 );
        $timeline->addItem(2, 'Event {id}', 'This is the event id: <b>{id}</b>
name: <b>{name}</b>', '2017-12-11 12:02:00', 'fa:arrow-left bg-green', 'left',
$obj2 );
        $timeline->addItem(3, 'Event {id}', 'This is the event id: <b>{id}</b>
name: <b>{name}</b>', '2017-12-13 12:03:00', 'fa:arrow-right bg-blue', 'right',
$obj3 );

        $timeline->setUseBothSides();
        $timeline->setTimeDisplayMask('dd/mm/yyyy');
        $timeline->setFinalIcon( 'fa:flag-checkered bg-red' );

        $display_condition = function( $object = false ) {
            if( in_array($object->id, [2,3]))
            {
                return true;
            }
            return false;
        };

        // adiciona uma ação à timeline
        $action1 = new TAction([<?this, 'onAction1'], ['id' => '{id}', 'name' =>
'{name'}]);
        $action2 = new TAction([<?this, 'onAction2'], ['id' => '{id}', 'name' =>
'{name'}]);

        $action1->setProperty('btn-class', 'btn btn-primary');
        $action2->setProperty('btn-class', 'btn btn-danger');

        $timeline->addAction($action1, 'Action 1', 'fa:bus', $display_condition );
        $timeline->addAction($action2, 'Action 2', 'fa:bus');

        parent::__construct($timeline);
    }
}
```

Os métodos a seguir foram programados para exibir uma mensagem na tela quando o usuário clicar nos botões de ação da timeline.

```
public static function onAction1($param)
{
    new TMessage('info', 'Action1 on Event ' . '<b>' . $param['id'] . '-' .
$param['name'] . '</b>');
}

public static function onAction2($param)
{
    new TMessage('info', 'Action2 on Event ' . '<b>' . $param['id'] . '-' .
$param['name'] . '</b>');
}
```

A figura a seguir demonstra o resultado da execução deste programa.

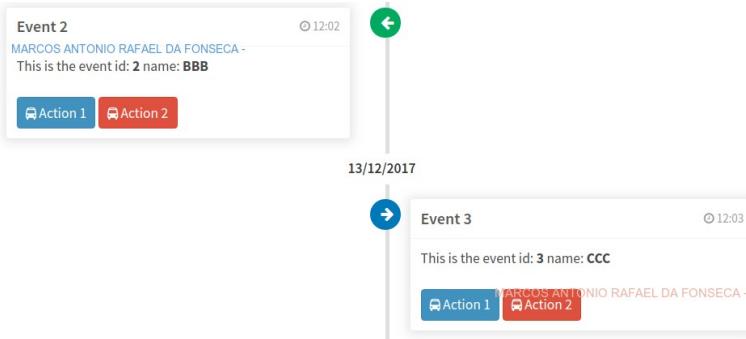


Figura 86 Timeline

4.8.4 Visão de cards

Na maioria dos exemplos em que precisamos exibir uma coleção de registros para o usuário, acabamos optando pela utilização de datagrids. Mas uma forma alternativa e bastante agradável para ser utilizada são os cards. A visão de cards apresenta os registros em blocos. Cada bloco pode conter um texto com vários atributos do objeto. Cada bloco também pode ter ações. Um exemplo comum para o uso de cards é a exibição de uma galeria de produtos.

Para criar uma visão de cards, utilizaremos o componente **TCardView**. Ele funciona de maneira bastante similar ao **TDataGrid**, visto que também utiliza o método **addItem()** para receber um objeto de dados, que serão utilizados para a montagem posterior dos cards. Cada objeto de dados adicionado pelo **addItem()** se transformará em um card.

O método **setTitleAttribute()** é utilizado para definir qual atributo do objeto de dados será utilizado como título do card. Já o método **setColorAttribute()** é usado para definir qual atributo definirá a cor do card (seu uso é opcional). O método **setItemTemplate()** é utilizado para definir um template para o conteúdo do card. Como parâmetro, podem ser informados quaisquer atributos do objeto de dados.

O método **addAction()** é usado para adicionar botões de ação aos cards. Ele recebe a ação em si (objeto **TAction**), o rótulo da ação, e o ícone da ação. No momento de instanciarmos a ação, devemos indicar os atributos a serem passados como parâmetro. Cada ação será apresentada na forma de um botão na parte inferior do card.

app/control/Presentation/Widgets/CardView.class.php

```
<?php
class CardView extends TPage
{
    public function __construct()
    {
        parent::__construct();
    }
}
```

```

// cria a visão de cards
$cards = new TCardView;

// adiciona objetos de dados aos cards
$items = [];
$items[] = (object) [
    'id' => 1, 'title' => 'item 1',
    'content' => 'item 1 content', 'color' => '#57D557'];
$items[] = (object) [
    'id' => 2, 'title' => 'item 2',
    'content' => 'item 2 content', 'color' => '#57D557'];
$items[] = (object) [
    'id' => 3, 'title' => 'item 3',
    'content' => 'item 3 content', 'color' => '#5950F1'];
$items[] = (object) [
    'id' => 4, 'title' => 'item 4',
    'content' => 'item 4 content', 'color' => '#57D557'];
$items[] = (object) [
    'id' => 5, 'title' => 'item 5',
    'content' => 'item 5 content', 'color' => '#CC2EC9'];
$items[] = (object) [
    'id' => 6, 'title' => 'item 6',
    'content' => 'item 6 content', 'color' => '#5950F1'];

foreach ($items as $key => $item)
{
    $cards->addItem($item);
}

// define os atributos para título e cor
$cards->setTitleAttribute('title');
$cards->setColorAttribute('color');

// define o template do card
$cards->setItemTemplate('<b>Content</b>: {content}');

// adiciona as ações
$edit_action = new TAction([$this, 'onItemEdit'], ['id'=> '{id}']);
$delete_action = new TAction([$this, 'onItemDelete'], ['id'=> '{id}']);

$cards->addAction($edit_action, 'Edit', 'fa:edit blue');
$cards->addAction($delete_action, 'Delete', 'fas:trash-alt red');

parent::add($cards);
}

```

Neste exemplo, criamos dois botões de ação: um para editar, e outro para excluir o item. A ação de edição está vinculada ao método `onItemEdit()`, que recebe um vetor de parâmetros. Neste vetor (`$param`) é possível ler o `id` do registro. O mesmo ocorre com o método `onItemDelete()`, executado sempre que o usuário clicar sobre o botão de excluir registro. Em ambos os casos, somente uma mensagem é apresentada ao usuário para demonstrar o funcionamento da ação.

```

public static function onItemEdit($param = NULL)
{
    new TMessage('info', '<b>onItemEdit</b><br>'.
        str_replace(',', ', <br>', json_encode($param)));
}

public static function onItemDelete($param = NULL)
{
    new TMessage('info', '<b>onItemDelete</b><br>'.
        str_replace(',', ', <br>', json_encode($param)));
}

```

A figura a seguir demonstra o resultado da execução deste programa.

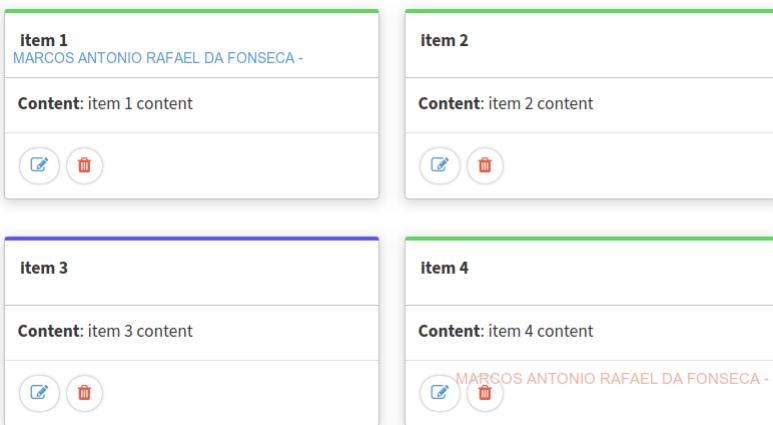


Figura 87 Cards

4.8.5 Kanban

Kanban é uma metodologia que representa o fluxo de produção. Muito utilizado como ferramenta de gerenciamento de projetos, o Kanban, divide a área visível em colunas, cuja sequencia determina a ordem de um processo (Ex. Análise, produto, desenvolvimento, testes, entrega), mas também pode simplesmente representar dias da semana. Dentro de cada coluna, são dispostos cards. Um card pode representar uma atividade que está em andamento. Esta atividade precisa progredir entre as colunas até ser concluída. Então, um kanban eletrônico, deve permitir mover a atividade entre as colunas.

Nosso componente para Kanban chama-se `TKanban`, e ele oferece vários recursos como: mover as colunas, mover os itens de uma coluna dentro dela, mover itens entre colunas, adicionar botões de ação na coluna, e no item, dentre outros.

O componente `TKanban`, inicia com a adição de colunas, ou fases, pelo método `addStage()`. Este método recebe o id da fase, e o seu nome. Em seguida, utilizamos o método `addItem()` para adicionar uma atividade em uma fase. Este método recebe o id da atividade, o id da fase, o título, o conteúdo, e uma cor.

Em seguida, precisamos definir uma série de ações que serão executadas em certos eventos. O método `addStageAction()`, por exemplo, adiciona uma ação na fase (coluna). Esta opção será exibida no canto superior direito da fase. Já o método `addItemAction()` adiciona um botão de ação no item. Este botão aparecerá na parte inferior do item.

Já o método `setItemDropAction()` define a ação a ser executada quando um item (atividade) for movida. Normalmente neste caso, é necessário atualizar o registro na base de dados. O método `setStageDropAction()` define a ação a ser executada quando a fase (coluna) for movida (alterada de posição).

app/control/Presentation/Widgets/KanbanView.class.php

```
<?php
class KanbanView extends TPage
{
    private $form;

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

        $kanban = new TKanban;

        $stages = [];
        $stages[] = [ 'id' => 1, 'title'=> 'stage 1'];
        $stages[] = [ 'id' => 2, 'title'=> 'stage 2'];
        $stages[] = [ 'id' => 3, 'title'=> 'stage 3'];

        foreach ($stages as $stage)
        {
            $kanban->addStage($stage['id'], $stage['title']);
        }

        $items = [];
        $items[] = [ 'id' => 101, 'stage_id' => 1, 'title' => 'item 1.1',
                    'content' => 'item 1.1 content', 'color' => '#FF1818'];
        $items[] = [ 'id' => 102, 'stage_id' => 1, 'title' => 'item 1.2',
                    'content' => 'item 1.2 content', 'color' => '#57D557'];
        $items[] = [ 'id' => 201, 'stage_id' => 2, 'title' => 'item 2.1',
                    'content' => 'item 2.1 content', 'color' => '#5950F1'];
        $items[] = [ 'id' => 202, 'stage_id' => 2, 'title' => 'item 2.2',
                    'content' => 'item 2.2 content', 'color' => '#57D557'];
        ...
        foreach ($items as $key => $item)
        {
            $kanban->addItem($item['id'], $item['stage_id'], $item['title'],
                $item['content'], $item['color']);
        }

        // adiciona ações na fase (coluna)
        $kanban->addStageAction('Action 1', new TAction([$this, 'onEditStage']),
            'fa:edit blue fa-fw');
        $kanban->addStageAction('Action 2', new TAction([$this, 'onDeleteStage']),
            'fa:trash red fa-fw');

        // adiciona ações no item
        $kanban->addItemAction('Edit', new TAction([$this, 'onItemEdit']),
            'fa:edit blue');
        $kanban->addItemAction('Delete', new TAction([$this, 'onItemDelete']),
            'fa:trash red');

        // define ações para os eventos de mover item e mover fase
        $kanban->setItemDropAction(new TAction([$this, 'onUpdateItemDrop']));
        $kanban->setStageDropAction(new TAction([$this, 'onUpdateStageDrop']));

        parent::__add($kanban);
    }
}
```

O método a seguir, é acionado quando a fase for movida (alterada de posição).

```
public static function onUpdateStageDrop($param)
{
    new TMessage('info', '<b>onUpdateStageDrop</b><br>' .
        str_replace(',', ', <br>', json_encode($param)));
}
```

O método a seguir é acionado quando um item for alterado de lugar.

```
public static function onUpdateItemDrop($param)
{
    new TMessage('info', '<b>onUpdateItemDrop</b><br>' .
        str_replace(',', '<br>', json_encode($param)));
}
```

O método a seguir é acionado quando o usuário clicar no botão para editar uma fase.

```
public static function onEditStage($param = NULL)
{
    new TMessage('info', '<b>onEditStage</b><br>' .
        str_replace(',', '<br>', json_encode($param)));
}
```

O método a seguir é acionado quando o usuário clicar no botão para excluir uma fase.

```
public static function onDeleteStage($param = NULL)
{
    new TMessage('info', '<b>onDeleteStage</b><br>' .
        str_replace(',', '<br>', json_encode($param)));
}
```

O método a seguir é acionado quando o usuário clicar no botão para editar um item.

```
public static function onItemEdit($param = NULL)
{
    new TMessage('info', '<b>onItemEdit</b><br>' .
        str_replace(',', '<br>', json_encode($param)));
}
```

O método a seguir é acionado quando o usuário clicar no botão para excluir um item.

```
public static function onItemDelete($param = NULL)
{
    new TMessage('info', '<b>onItemDelete</b><br>' .
        str_replace(',', '<br>', json_encode($param)));
}
```

A figura a seguir demonstra o resultado da execução deste programa.

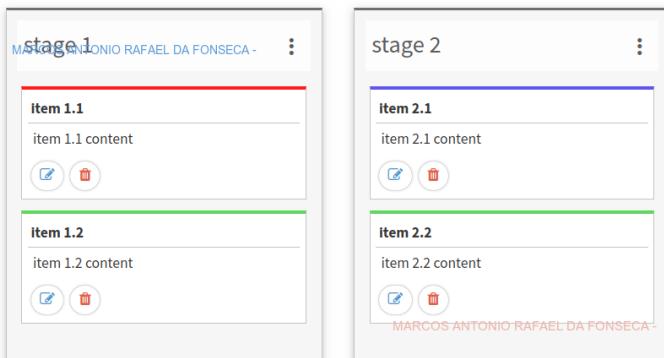


Figura 88 Kanban

4.8.6 Árvore

O objetivo do componente `TTreeView` é oferecer ao usuário uma visão de árvore com vários níveis. Este componente pode oferecer ao usuário uma forma de selecionar informações hierárquicas, como categorias de um produto, por exemplo.

Para utilizar um `TTreeView` é bastante simples. Inicialmente devemos criar um vetor multidimensional com a estrutura da árvore. Neste exemplo, o vetor `$data` representa a estrutura da árvore. Esta estrutura poderá ter muitos níveis. O último nível do vetor será utilizado para formar os nodos folha da árvore (neste caso, os códigos e os nomes das cidades). Para preencher uma árvore, basta utilizar o método `fromArray()` da classe `TTreeView`. Ainda podemos utilizar os métodos `setSize()` para definir a sua largura, e `setItemAction()` para definir a ação a ser executada quando o usuário clicar em nodo da árvore.

Neste exemplo, além da árvore, também é criado um formulário (`BootstrapFormBuilder`) para demonstrar a seleção de um item da árvore. Este formulário terá apenas dois campos: Chave (`key`) e Valor (`value`). Tanto a árvore, quanto o formulário, serão colocados lado a lado dentro da página por meio de uma caixa (`THBox`).

app/control/Presentation/Widgets/TreeView.class.php

```
<?php
class FormTreeView extends TPage
{
    private $form;
    function __construct()
    {
        parent::__construct();

        // define os dados da árvore
        $data = array();
        $data['Brazil']['RS'][10] = 'Lajeado';
        $data['Brazil']['RS'][20] = 'Cruzeiro do Sul';
        $data['Brazil']['RS'][30] = 'Porto Alegre';
        $data['Brazil']['SP'][40] = 'São Paulo';
        $data['Brazil']['SP'][50] = 'Osasco';
        $data['Brazil']['MG'][60] = 'Belo Horizonte';
        $data['Brazil']['MG'][70] = 'Ipatinga';

        // cria uma scroll ao redor da árvore
        $scroll = new TScroll;
        $scroll->setSize(300, 200);

        // cria a árvore
        $treeview = new TTreeView;
        $treeview->setSize(300); // tamanho
        $treeview->setItemAction(new TAction(array($this, 'onSelect')));
        $treeview->fromArray($data); // preenche a árvore
        $scroll->add($treeview); // coloca a árvore na scroll

        // Cria o formulário
        $this->form = new BootstrapFormBuilder('form_test');
        $this->form->setFormTitle('Form');

        // cria os campos
        $key = new TEntry('key');
        $value = new TEntry('value');

        // adiciona os campos na scroll
        $scroll->add($key);
        $scroll->add($value);
    }

    // função que é chamada quando o usuário clica no item
    function onSelect(TAction $action)
    {
        $item = $action->getItem();
        $value = $item->getValue();
        echo "Você selecionou: " . $value;
    }
}
```

```

    // adiciona os campos ao formulário
    $this->form->addFields( [new TLabel('Key')], [{$key} ] );
    $this->form->addFields( [new TLabel('Value')], [{$value} ] );

    // empacota scroll e formulário lado a lado
    $hbox = THBox::pack($scroll, $this->form);
    $hbox->style = 'display:inline-flex';

    // adiciona a caixa horizontal à página
    parent::add($hbox);
}

```

Quando o usuário clicar sobre algum nodo folha da árvore, o método `onSelect()` será executado. A execução deste método foi programada pelo `setItemAction()` da `TTreeView`. Toda a ação programada pelo `setItemAction()`, recebe por padrão um vetor contendo duas posições: `key`, contendo o índice do nodo selecionado, e `value`, contendo o valor. Neste exemplo, utilizamos os valores selecionados para enviar ao formulário por meio do método `sendData()`, sendo necessário saber o nome do formulário em questão (`form_test`), tendo em vista que esta é uma execução estática.

```

public static function onSelect($param)
{
    // monta objeto para enviar dados ao formulário
    $obj = new StdClass;
    $obj->key = $param['key']; // obtém a chave do nodo (índice)
    $obj->value = $param['value']; // obtém o valor do nodo (conteúdo)

    // preenche o formulário com os atributos do objeto
    TForm:::sendData('form_test', $obj);
}

```

Na próxima figura, podemos ver o resultado da execução deste programa.

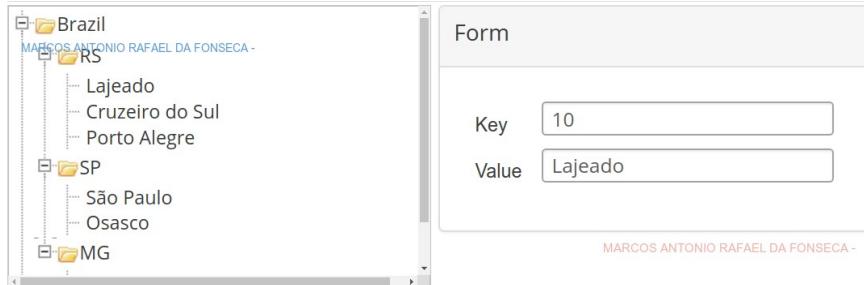


Figura 89 Componente de árvore

4.8.7 Calendário

Um componente bastante requisitado é o calendário. O calendário pode ser utilizado para diversas finalidades como o agendamento de serviços, atendimentos, contatos. Pode ser utilizado na área de prestação de serviços, de vendas, de planejamento, dentre outros. O calendário é implementado pelo componente `TFullCalendar`.

Ao instanciarmos o calendário (`TFullCalendar`), identificamos a data inicial, e a forma de visualização: mensal (`month`), semanal (`agendaWeek`), ou diária (`agendaDay`). O componente ainda oferece o método `setTimeRange()`, que permite definir um intervalo de horário visível no calendário.

O calendário funciona com base na adição de eventos. Cada evento possui um id, título, início, fim, etc. Para demonstrar o funcionamento do calendário, criaremos uma série de eventos fictícios, baseados na data atual. Fizemos isso para que o exemplo sempre apresente eventos dentro da semana atual sendo vista pelo usuário. Por isso declaramos algumas variáveis como `$today` (hoje), `$before_yesterday` (anteontem), `$yesterday` (ontem), `$tomorrow` (amanhã), e `$after_tomorrow` (depois de amanhã). Essas variáveis são calculadas pela classe `DateTime` do PHP.

Após calcular as variáveis de tempo dinamicamente, usamos o método `addEvent()` para adicionar alguns eventos ao calendário. Os eventos variarão entre anteontem, e depois de amanhã, com diferentes horários. O método `addEvent()` recebe como parâmetros um id, título, início, fim, e opcionalmente uma URL e uma cor.

Após adicionarmos alguns eventos, utilizamos dois métodos para programar eventos de clique no calendário. O método `setDayClickAction()` programa uma ação para ser executada quando o usuário clicar em um espaço vazio, ainda sem evento. Já o método `setEventClickAction()` programa uma ação para ser executada quando o usuário clicar em um evento já registrado.

Quando temos eventos cadastrados a partir do banco de dados, o método `setDayClickAction()` pode ser usado para acionar um formulário de cadastro de novo evento, e o método `setEventClickAction()` pode ser usado para acionar um formulário de edição de evento.

app/control/Presentation/Widgets/FullCalendarStaticView.class.php

```
<?php
class FullCalendarStaticView extends TPage
{
    private $fc;

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

        // instancia o calendário
        $this->fc = new TFullCalendar(date('Y-m-d'), 'month');
        $this->fc->setTimeRange( '06:00:00', '20:00:00' );

        $today = date("Y-m-d");// dia atual
        $before_yesterday = (new DateTime(date('Y-m-d')))->
            sub(new DateInterval("P2D"))->format('Y-m-d');
        $yesterday = (new DateTime(date('Y-m-d')))->
            sub(new DateInterval("P1D"))->format('Y-m-d');
        $tomorrow = (new DateTime(date('Y-m-d')))->
            add(new DateInterval("P1D"))->format('Y-m-d');
        $after_tomorrow = (new DateTime(date('Y-m-d')))->
            add(new DateInterval("P2D"))->format('Y-m-d');
    }
}
```

```

// adiciona eventos ao calendário
$this->fc->addEvent(1, 'Event 1', $before_yesterday.'T08:30:00',
                      $before_yesterday.'T12:30:00', null, '#C04747');
$this->fc->addEvent(2, 'Event 2', $before_yesterday.'T14:30:00',
                      $before_yesterday.'T18:30:00', null, '#668BC6');
$this->fc->addEvent(3, 'Event 3', $yesterday.'T08:30:00',
                      $yesterday.'T12:30:00', null, '#C04747');
$this->fc->addEvent(4, 'Event 4', $yesterday.'T14:30:00',
                      $yesterday.'T18:30:00', null, '#668BC6');
$this->fc->addEvent(5, 'Event 5', $today.'T08:30:00',
                      $today.'T12:30:00', null, '#FF0000');
$this->fc->addEvent(6, 'Event 6', $today.'T14:30:00',
                      $today.'T18:30:00', null, '#5AB34B');
$this->fc->addEvent(7, 'Event 7', $tomorrow.'T08:30:00',
                      $tomorrow.'T12:30:00', null, '#FF0000');
$this->fc->addEvent(8, 'Event 8', $tomorrow.'T14:30:00',
                      $tomorrow.'T18:30:00', null, '#5AB34B');
$this->fc->addEvent(9, 'Event 9', $after_tomorrow.'T08:30:00',
                      $after_tomorrow.'T12:30:00', null, '#FF0000');
$this->fc->addEvent(10, 'Event 10', $after_tomorrow.'T14:30:00',
                      $after_tomorrow.'T18:30:00', null, '#FF8C05');

// programa ação para clique em espaço vazio
$this->fc->setDayClickAction(new TAction(array($this, 'onDayClick')));

// programa ação para clique em evento
$this->fc->setEventClickAction(new TAction(array($this, 'onEventClick')));
parent::add( $this->fc );
}

```

O método `onDayClick()` será executado sempre que o usuário clicar em um espaço ainda sem evento registrado. Neste caso, ele exibirá a data/hora correspondente ao slot clicado. Quando integrado ao banco de dados, esta ação pode acionar um formulário vazio de cadastro de novo evento.

```

public static function onDayClick($param)
{
    $date = $param['date'];
    new TMessage('info', "You clicked at date: {$date}");
}

```

O método `onEventClick()` será executado sempre que o usuário clicar em um evento já registrado no calendário (pelo `addEvent`). Neste caso, ele exibirá o identificador do evento clicado. Quando integrado ao banco de dados, esta ação pode acionar um formulário de edição de evento.

```

public static function onEventClick($param)
{
    $id = $param['id'];
    new TMessage('info', "You clicked at id: {$id}");
}

```

Na próxima figura, podemos ver o resultado da execução deste programa.

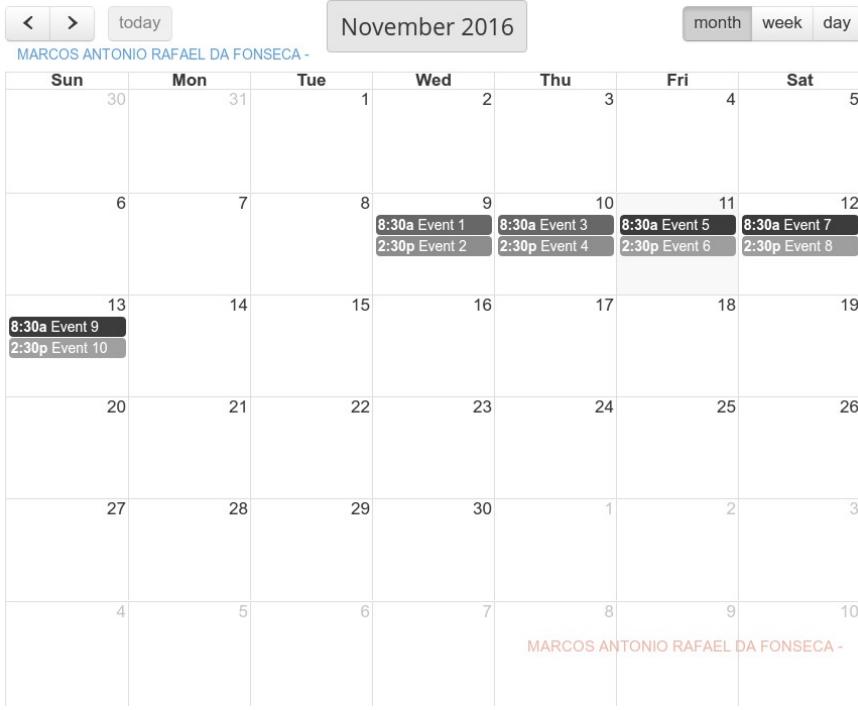


Figura 90 Calendário

4.9 Templates e novos componentes

Nessa seção veremos como criar formas de apresentação mais livres para as páginas, por meio de templates HTML, e também por meio de novos componentes.

4.9.1 Template View básico

Por mais que possamos criar componentes de software, existem limitações que devem ser reconhecidas. Não seria prudente nem inteligente construir um framework com milhares de componentes, imaginando todas as necessidades que diferentes projetos possuem. Um Framework é um trabalho inacabado, um arcabouço de software que atenderá talvez 80% das necessidades de um projeto. Sempre que uma necessidade específica não coberta pelo Framework surgir, devemos ter alternativas definidas para sua implementação. No caso do Adianti Framework podemos criar um componente filho de `TElement` em `app/lib`, ou utilizar um Template View.

Partimos para a criação de componentes de software quando a necessidade irá se repetir, tem características claras e uma interface bem definida. Mas às vezes, uma determinada página tem uma necessidade específica de apresentação que não justifica a criação de um componente. Neste caso, podemos utilizar um Template View.

Um Template View é um padrão de projeto que permite uma flexibilidade na apresentação da aplicação por meio do uso de HTML. A ideia do Template View é a apresentação de HTML com pequenas marcações em seu conteúdo, sendo que essas marcações são substituídas por conteúdo dinâmico definido em nível de aplicação.

A implementação do padrão Template View no Adianti Framework é realizada pela classe `THtmlRenderer`. O primeiro passo para a utilização dessa classe é a definição de um template. Nesse caso em específico, desejamos exibir algumas informações do cliente em uma seção e as informações dos seus contatos (de maneira repetitiva) em outra seção, como pode ser conferido na figura a seguir.

The screenshot shows a web page with a header "Customer data" and a sub-header "MARCOS ANTONIO RAFAEL DA FONSECA". Below this, there are two sections: "Name" (Andrei Zmievski) and "Address" (Rua Palo Alto). A link "Click here to edit the customer" is present. The next section is titled "Contacts", which contains a table with columns "Type" and "Value". It lists three contacts: MSN (andrei@msn.com), ICQ (6232071023124), and Gmail (andrei@gmail.com). The footer displays the name again: "MARCOS ANTONIO RAFAEL DA FONSECA -".

Figura 91 Template View com uma seção simples e uma repetitiva

Um template é um HTML com marcações que seguem um determinado padrão. Para o framework, todo template deve estar contido na seção `<!--[main]-->`. Além disso, temos marcações de início de seção `<!--[contacts]-->` e de final de seção `<!--[/contacts]-->`, além de marcação de substituição de variáveis como `{$name}` e `{$address}`. Neste exemplo, temos uma seção `contacts` e `contacts-detail`. Na seção `main`, exibiremos algumas informações básicas do cliente como nome `{$name}` e endereço `{$address}`. Já a seção `contacts` engloba uma linha de título (Contacts), linha de cabeçalho e uma seção interna (`contacts-detail`), que será responsável por apresentar pares de valores (`$type` e `$value`), de maneira repetitiva.

`app/resources/customer.html`

```
<!--[main]-->
<link href="app/resources/styles.css" rel="stylesheet" type="text/css" media="screen" />
<table class="customform">
    <tr>
        <td colspan="2" class="formtitle">Customer data</div></td>
    </tr>
    <tr>
        <td width="50%><b>Name</b></td>
        <td width="50%"><span class="formfield">{$name}</span></td>
    </tr>
    <tr>
        <td><b>Address</b></td>
        <td><span class="formfield">{$address}</span></td>
    </tr>
    <tr>
        <td colspan="2" align="center">
            <a generator="adiantti" href="index.php?class=CustomerFormView&method=onEdit

```

```

        </td>
    </tr>
    <!--[contacts]-->
    <tr>
        <td class="sectiontitle" colspan=2>Contacts</td>
    </tr>
    <tr bgcolor="#e0e0e0">
        <td>Type</td>
        <td>Value</td>
    </tr>
    <!--[contacts-detail]-->
    <tr>
        <td>{$type}</td>
        <td><i>{$value}</i></td>
    </tr>
    <!--[/contacts-detail]-->

    <!--[/contacts]-->
</table>
<!--[/main]-->
```

Agora que já temos o template definido, partimos para a construção do código que irá utilizá-lo, na classe `TemplateViewBasicView`. Para tal, criamos um objeto da classe `THtmlRenderer`. Este objeto será responsável por interpretar o HTML e realizar as substituições necessárias. No Adianti Framework temos algumas operações básicas que podem ser realizadas em um template que são: a exibição ou não de uma seção; a repetição de uma seção; e a substituição de variáveis de uma seção.

Nesse exemplo, estamos abrindo uma transação com a base de dados e carregando o cliente 1. A partir de então, definimos um vetor de substituições (`$replace`), contendo o nome variável da marcação em seu índice; e o conteúdo substituto como valor do vetor. Assim, o índice do vetor `code`, representará no Template a variável `${code}`. O método `enableSection()` habilita uma seção do Template e ao mesmo tempo indica um vetor de substituições para aquela seção.

Logo após habilitarmos a seção principal do documento, percorremos os contatos de um cliente por meio do método `getContacts()` e formamos uma nova matriz de substituições, onde cada linha contém um par de substituições (`type` e `value`). Em seguida, habilitamos a seção `contacts` e a seção `contacts-detail`. Ao habilitarmos a seção `contacts-detail`, além de informarmos o vetor de substituições, informamos um terceiro parâmetro, que quando é verdadeiro (`TRUE`), indica que a seção é repetitiva. Sendo assim, a seção `contacts-detail` será repetida conforme a quantidade de linhas da matriz de substituições (`$replace`). Para cada linha da matriz, um bloco daquela seção do Template, que contém uma linha da tabela, será criada. Por fim, acrescentamos o objeto `THtmlRenderer` à página.

app/control/Presentation/Extending/TemplateViewBasicView.class.php

```
<?php
class TemplateViewBasicView extends TPage
{
    public function __construct()
    {
        parent::__construct();

        // carrega o template
        $this->html = new THtmlRenderer('app/resources/customer.html');
        try {
            // Abre transação com a base de dados
            TTransaction::open('samples');

            // carrega o cliente 1 em memória
            $customer = new Customer(1);

            // define as substituições para a seção main
            $replace = array();
            $replace['code'] = $customer->id;
            $replace['name'] = $customer->name;
            $replace['address'] = $customer->address;

            // habilita a seção main com as substituições
            $this->html->enableSection('main', $replace);

            // define as substituições, baseado nos contatos
            $replace = array();
            foreach ($customer->getContacts() as $contact)
            {
                $replace[] = array('type' => $contact->type,
                                  'value'=> $contact->value);
            }

            // habilita as seções contacts e contacts-detail (repetitiva)
            $this->html->enableSection('contacts');
            $this->html->enableSection('contacts-detail', $replace, TRUE);

            parent::add($this->html);
            TTransaction::close();
        }
        catch (Exception $e)
        {
            new TMessage('error', $e->getMessage());
        }
    }
}
```

4.9.2 Template View avançado

Agora que já conhecemos as características básicas de utilização da classe `THtmlRenderer` no exemplo anterior, podemos criar um exemplo um pouco mais elaborado. Este próximo exemplo terá duas seções: a principal, chamada `main`; e uma outra chamada `object`. Na seção principal, faremos algumas substituições simples de variáveis com dados quaisquer. Já na seção `object`, colocaremos um objeto construído pelo Framework. Dessa forma, a variável `[$widget]` não será substituída simplesmente por um valor escalar (integer, string, float), mas por um objeto instanciado a partir de um componente visual do Framework, como um `TForm`, ou `TDataSet`.

app/resources/content.html

```

<!--[main]-->
<link href="app/resources/styles.css" rel="stylesheet" type="text/css" media="screen" />

<table class="customform">
    <tr>
        <td colspan="2" class="formtitle">Customer data</td>
    </tr>
    <tr>
        <td width="50%">Name</td>
        <td width="50%">{$name}</td>
    </tr>
    <tr>
        <td>Address</td>
        <td>{$address}</td>
    </tr>
    <!--[object]-->
    <tr>
        <td class="sectiontitle" colspan=2>Embedded object</td>
    </tr>
    <tr bgcolor="#e0e0e0">
        <td colspan="2">A Framework object ({$class})</td>
    </tr>
    <tr>
        <td colspan="2">{$widget}</td>
    </tr>
    <!--[/object]-->
</table>
<!--[/main]-->

```

Além de renderizar o conteúdo do HTML, criaremos dois botões de ação: “Action 1” e “Action 2”. Estes botões de ação serão criados via programação e não pelo template. Na figura a seguir temos a tela inicial do programa, onde somente a seção `main` é exibida, sem o conteúdo da seção `object`. Os botões de ação são vistos no topo.



Figura 92 Template View com uma seção simples habilitada

O objetivo do botão “Action 1” é habilitar a seção `object`, e inserir nesta seção um formulário `BootstrapFormBuilder` no lugar da marcação `[$widget]`, como pode ser visto na figura a seguir.

The screenshot shows a template view with the following structure:

- Action Buttons:** Two buttons labeled "Action 1" and "Action 2".
- User Information:** Text "MARCOS ANTONIO RAFAEL DA FONSECA -" followed by "Customer data".
- Form Fields:**
 - Name: Test name
 - Address: Test address
- Section Header:** "Embedded object".
- Description:** "A Framework object (Adianti\Wrapper\BootstrapFormBuilder)".
- Form Fields:**

Test1	<input type="text"/>
Test2	<input type="text"/>
- Buttons:** A "Show" button with a radio icon.
- Text:** "MARCOS ANTONIO RAFAEL DA FONSECA -" at the bottom right.

Figura 93 Template View com uma seção contendo um formulário

Já o objetivo do botão “Action 2” é exibir a seção **object**, e inserir nela uma **TQuickGrid** no lugar da marcação **{\$widget}**, como pode ser visto na figura a seguir.

The screenshot shows a template view with the following structure:

- Action Buttons:** Two buttons labeled "Action 1" and "Action 2".
- User Information:** Text "MARCOS ANTONIO RAFAEL DA FONSECA -" followed by "Customer data".
- Form Fields:**
 - Name: Test name
 - Address: Test address
- Section Header:** "Embedded object".
- Description:** "A Framework object (Adianti\Wrapper\BootstrapDatagridWrapper)".
- Table:**

Code	Name
001	Test 001
002	Test 002
- Text:** "MARCOS ANTONIO RAFAEL DA FONSECA -" at the bottom right.

Figura 94 Template View com uma seção contendo umadatagrid

Agora que já entendemos como funcionará esta página, podemos estudar seu código-fonte. Esta página terá apenas dois botões de ação (**TActionLink**): “Action 1”, que estará conectado ao método **onAction1()**; e “Action 2”, que estará conectado ao método **onAction2()**. Cada uma dessas ações habilitará a seção **object** do template, inserindo um diferente objeto criado dinamicamente. Para cada botão criado, definimos sua classe (**btn-default**). Estes botões são encapsulados em uma caixa **THBox**.

Após criar o formulário, criamos um objeto **THtmlRenderer** a partir da leitura do template **content.html**. Em seguida, definimos um vetor (**\$replace**) de substituições para as variáveis da seção **main**. A seção é habilitada pelo método **enableSection()**, que além de habilitar a seção, recebe o vetor de substituições. Ao final, criamos um contêiner (**TVBox**) para empacotar os objetos.

app/control/Presentation/Extending/TemplateViewAdvancedView.class.php

```
<?php
class TemplateViewAdvancedView extends TPage
{
    private $quickform;
    public function __construct()
    {
        parent::__construct();

        // cria botões de ação direta (action link)
        $link1 = new TActionLink('Action 1', new TAction(array($this, 'onAction1')),
            'green', 10, null, 'fa:search');
        $link2 = new TActionLink('Action 2', new TAction(array($this, 'onAction2')),
            'blue', 10, null, 'fa:search');
        $link1->class = 'btn btn-default';
        $link2->class = 'btn btn-default';

        // empacota os botões em uma caixa horizontal
        $hbox_actions = THBox::pack($link1, $link2);

        try {
            // cria o renderizador de HTML
            $this->html = new THtmlRenderer('app/resources/content.html');

            // define substituições para a seção main
            $replace = array();
            $replace['name'] = 'Test name';
            $replace['address'] = 'Test address';

            // habilita e substitui as variáveis em main
            $this->html->enableSection('main', $replace);

            // cria um container para empacotar elementos
            $container = new TVBox;
            $container->style = 'width:100%';
            $container->add($hbox_actions);
            $container->add($this->html);
            parent::__add($container);
        }
        catch (Exception $e) {
            new TMessage('error', $e->getMessage());
        }
    }
}
```

O método `onAction1()` é executado quando o usuário clicar no botão "Action 1". Este botão cria um formulário utilizando a classe `BootstrapFormBuilder` e adiciona dois campos no mesmo, além de uma ação, conectada ao método `onForm1()`. Logo em seguida, é definido o vetor de substituições. O método `enableSection()` habilita a seção `object`, passando o vetor de substituições, que contém na posição `widget`, o próprio objeto da classe `BootstrapFormBuilder`.

```
public function onAction1()
{
    // cria o formulário
    $form = new BootstrapFormBuilder('form1');
    $form->addField([new TLabel('Test1')], [new TEntry('test1')]);
    $form->addField([new TLabel('Test2')], [new TEntry('test2')]);
    $form->addAction('Show', new TAction(array($this, 'onForm1')), 'fa:check-circle-o');
```

```

// define o vetor de substituições
$replace = array();
$replace['widget'] = $form;
$replace['class'] = get_class($form);

// habilita a seção object, passando o vetor de substituições
$this->html->enableSection('object', $replace);
}

```

O método `onForm1()` é executado em reação a uma ação desse formulário. Esse método simplesmente exibe os dados preenchidos no formulário.

```

public static function onForm1($param)
{
    new TMessage('info', json_encode($param));
}

```

O método `onAction2()` é executado quando o usuário clicar no botão "Action 2". Este método também habilita a seção `object` e em seguida passa um objeto do tipo `TQuickGrid`. Para tal, inicialmente a `TQuickGrid` é criada, são adicionadas duas colunas (`code` e `name`) e são adicionados dois objetos por meio do método `addItem()`.

Logo após adicionar as linhas na datagrid, é definido um vetor de substituições (`$replace`). Este vetor recebe o objeto `TQuickGrid` na posição `widget`. Ao final, a seção `object` é habilitada informando também o vetor de substituições.

```

public function onAction2()
{
    $datagrid = new BootstrapDatagridWrapper(new TQuickGrid);
    $datagrid->width = '100%';

    // adiciona as colunas
    $datagrid->addQuickColumn('Code',      'code',      'right', '30%');
    $datagrid->addQuickColumn('Name',      'name',      'left',   '70%');

    // cria o modelo em memória
    $datagrid->createModel();

    // adiciona uma linhas
    $object = new StdClass;
    $object->code = '001';
    $object->name = 'Test 001';
    $datagrid->addItem($object);

    // adiciona uma linhas
    $object = new StdClass;
    $object->code = '002';
    $object->name = 'Test 002';
    $datagrid->addItem($object);

    // define o vetor de substituições
    $replace = array();
    $replace['widget'] = $datagrid;
    $replace['class'] = get_class($datagrid);

    // habilita a seção, passando o vetor de substituições
    $this->html->enableSection('object', $replace);
}

```

4.9.3 Template View com matrizes

Nos exemplos anteriores, vimos como habilitar diferentes seções dentro de um documento com sucessivas chamadas do método `enableSection()`. Neste exemplo, veremos como é possível construir uma matriz em PHP com diversos níveis de dados, e realizar a substituição no Template de uma única vez.

Para iniciar, construiremos um Template com, pelo menos, três níveis de seções. Na seção `main`, teremos variáveis a serem substituídas como `{$object->id}` e `{$object->name}`. Na seção `accounts1`, teremos variáveis como `{$date}`, e `{$value}`. Dentro da seção `accounts1`, ainda teremos uma outra seção `details`, contendo variáveis como `{$product}`, e `{$value}`. Perceba que as seções `main`, `accounts1`, e `detail` formam uma três níveis de seções. Ainda teremos um bloco `accounts2`, igual à `accounts1`. Dentro do Template, é possível aplicar ainda as seguintes funções em variáveis:

Tabela 4. Funções de Template

Função	Descrição
<code>number_format({\$value},2,',','.'</code>)	Formata valores numéricos.
<code>date_format({\$date}, 'd/m/Y')</code>	Converte data para um formato específico.
<code>evaluate({\$qty} * {\$value})</code>	Resolve cálculos.

```
app/resources/customer_accounts.html
<!--[main]-->
<link href="app/resources/styles.css" rel="stylesheet" type="text/css" media="screen" />

<table class="customform" style="width:100%">
    <tr>
        <td colspan="4" class="formtitle">Customer data</td>
    </tr>
    <tr>
        <td width="50%"><b>Code</b></td>
        <td width="50%" colspan="3"><span class="formfield">{$object->id}</span></td>
    </tr>
    <tr>
        <td><b>Name</b></td>
        <td colspan="3"><span class="formfield">{$object->name}</span></td>
    </tr>
    <!--[header]-->
    <tr>
        <td><b>{$name}</b></td>
        <td colspan="3"><span class="formfield">{$value}</span></td>
    </tr>
    <!--[/header]-->

    <tr>
        <td class="sectiontitle" colspan="4">Detail 1</td>
    </tr>

    <!--[accounts1]-->
    <tr bgcolor="#00e0e0">
        <td align="center" colspan="4"><b>Date</b>: date_format({$date}, 'd/m/Y') - 
            <b>Total sale</b>: R$ number_format({$value},2,',','.'

```

```

<tr bgcolor="#FFF000">
    <td align="center"><b>Product</b></td>
    <td align="center"><b>Qty</b></td>
    <td align="center"><b>Value</b></td>
    <td align="center"><b>Total</b></td>
</tr>
<!--[details]-->
<tr>
    <td align="center">{$product}</td>
    <td align="center">{$qty}</td>
    <td align="center">R$ number_format({$value},2,',','.')</td>
    <td align="center">R$ number_format(evaluate({$qty} * {$value}), 2, ',', '.')</td>
</td>
</tr>
<!--[/details]-->
<!--[/accounts1]-->

<tr>
    <td class="sectiontitle" colspan="4">Detail 2</td>
</tr>
<!--[accounts2]-->
<tr bgcolor="#00e0e0">
    <td align="center" colspan="4"><b>Date</b>: date_format({$date}, 'd/m/Y') - 
        <b>Total sale</b>: R$ number_format({$value}, 2, ',', '.')</td>
</td>
</tr>
<tr bgcolor="#FFF000">
    <td align="center"><b>Product</b></td>
    <td align="center"><b>Qty</b></td>
    <td align="center"><b>Value</b></td>
    <td align="center"><b>Total</b></td>
</tr>
<!--[details]-->
<tr>
    <td align="center">{$product}</td>
    <td align="center">{$qty}</td>
    <td align="center">R$ number_format({$value}, 2, ',', '.')</td>
    <td align="center">R$ number_format(evaluate({$qty} * {$value}), 2, ',', '.')</td>
</td>
</tr>
<!--[/details]-->
<!--[/accounts2]-->
</table>
<!--[/main]-->

```

Agora, construiremos o programa que realizará o preenchimento das seções, inclusive das repetições, por meio de uma única matriz de dados. O exemplo inicia com a instância de `THtmlRenderer`. Logo em seguida, começamos a definir a matriz de substituições (`$replace`). Já na primeira posição (`object`), a matriz recebe um objeto (`$plain_object`). Isto permitirá que de dentro do Template, seus atributos (Ex: `[$object->id]`) possam ser acessados diretamente. Em seguida, definimos o conteúdo da seção `header`. Note que cada seção é encapsulada pela notação de vetor `[]`, tendo em vista que ela poderá conter repetições. Mesmo não havendo repetições, cada seção é contida por um vetor.

Em seguida, criamos a posição `accounts1`. Esta seção terá 2 blocos: a venda do dia 2016-05-01, e a venda do dia 2016-05-03, cada um sendo uma matriz dentro da matriz. Poderíamos ter vários blocos ali, cada um representando uma venda. Perceba que o subnível “`details`”, ocorre 2 vezes, uma dentro de cada bloco. Dentro do bloco

“details” temos outro nível de matriz, contendo 2 itens (produtos). Ali, também poderíamos ter vários outros produtos representados, bastando repetir o bloco interior do nível “details”, respeitando a sintaxe de declaração de vetores do PHP.

app/control/Presentation/Extending/TemplateRepeatView.class.php

```
<?php
class TemplateRepeatView extends TPage
{
    public function __construct()
    {
        parent::__construct();

        // cria o renderizador
        $this->html = new THtmlRenderer('app/resources/customer_accounts.html');

        try
        {
            // cria um objeto simples para substituição
            $plain_object = new stdClass;
            $plain_object->id    = '001';
            $plain_object->name = 'John';

            // define as substituições
            $replace = array();
            $replace['object']      = $plain_object;

            $replace['header']     = [ [ 'name' => 'Field test',
                                         'value' => 'Field value' ] ];

            $replace['accounts1'] = [ [ [ 'date'=>'2016-05-01',
                                         'value'=>100,
                                         'details' => [ [ 'product'=> 'Chocolate',
                                                         'qty'=> 10,
                                                         'value' => 5 ],
                                                         [ 'product'=> 'Milk',
                                                         'qty'=> 5,
                                                         'value' => 10 ] ] ],
                                     [ 'date' => '2016-05-03',
                                       'value' => 200,
                                       'details' => [ [ 'product' => 'Cofee',
                                                       'qty' => 10,
                                                       'value' => 10 ],
                                                       [ 'product'=> 'Pizza',
                                                         'qty' => 5,
                                                         'value' => 20 ] ] ] ];
            $replace['accounts2'] = [ ... ];

            // habilita a seção main, passando a matriz de substituições
            $this->html->enableSection('main', $replace);

            parent::add($this->html);
        }
        catch (Exception $e)
        {
            new TMessage('error', $e->getMessage());
        }
    }
}
```

Na próxima figura, podemos ver o resultado da execução deste programa.

Customer data						
MARCOS ANTONIO RAFAEL DA FONSECA -						
Code	001					
Name	John					
Field test	Field value					
Detail 1						
Date: 01/05/2016 - Total sale: R\$ 100,00						
Product	Qty	Value	Total			
Chocolate	10	R\$ 5,00	R\$ 50,00			
Milk	5	R\$ 10,00	R\$ 50,00			
Date: 03/05/2016 - Total sale: R\$ 200,00						
Product	Qty	Value	Total			
Cofee	10	R\$ 10,00	R\$ 100,00			
Pizza	5	R\$ 20,00	R\$ 100,00			
Detail 2						
Date: 07/05/2016 - Total sale: R\$ 100,00						
Product	Qty	Value	Total			
Pendrive	10	R\$ 5,00	R\$ 50,00			
SDCard	5	R\$ 10,00	R\$ 50,00			
Date: 08/05/2016 - Total sale: R\$ 200,00						
Product	Qty	Value	Total			
DVD-R	10	R\$ 10,00	R\$ 100,00			
Mouse	5	R\$ 20,00	R\$ 100,00			

Figura 95 Template baseado em matrizes

4.9.4 Criando componentes

Mesmo que o Framework ofereça uma série de componentes, chega o momento em que precisamos de algo a mais. Nesta situação, é importante que saibamos estender o Framework, acrescentando a ele novos componentes. O objetivo deste exemplo é criar um componente para implementar um Accordion. Um Accordion divide a tela em várias porções, criando um efeito “acordeão”, com botões que podem ser clicados, expandindo uma área, até então não visível. Para criar um elemento, na maioria das vezes podemos estender a classe `TElement`, que é base para todos os elementos HTML.

No método construtor, chamamos o construtor da classe pai, passando 'div' como parâmetro, dizendo que o elemento que desejamos construir é um `<div>`. Nesse momento também definimos o ID do elemento e inicializamos o vetor `elements`, que armazenará os elementos do Accordion.

O método `appendPage()` será responsável por adicionar um objeto ao Accordion. Para tal, ele recebe o título, e o próprio objeto a ser adicionado. Já o método `show()` é responsável por exibir o conteúdo do componente. Para tal, ele percorre os elementos-filho (`elements`), um a um, adicionando-os no elemento pai. Em seguida, precisamos importar o Javascript e o CSS que irão respectivamente “ligar” o accordion e apresentar ele da maneira correta em tela.

app/lib/widget/TAccordion.class.php

```

<?php
class TAccordion extends TElement
{
    protected $elements;

    public function __construct()
    {
        parent::__construct('div');
        $this->id = 'taccordion' . uniqid();
        $this->elements = array();
    }

    /**
     * Método para adicionar uma “página” de conteúdo ao accordion
     */
    public function appendPage($title, $object)
    {
        $this->elements[] = array($title, $object);
    }

    public function show()
    {
        foreach ($this->elements as $child)
        {
            $title = new TElement('button');
            $title->class = 'taccordion';
            $title->add($child[0]);

            $content = new TElement('div');
            $content->class = 'taccordion-content';
            $content->add($child[1]);

            parent::add($title);
            parent::add($content);
        }
    }

    TStyle::importFromFile('app/lib/include/taccordion/taccordion.css');
    TScript::importFromFile('app/lib/include/taccordion/taccordion.js');

    parent::show();
}
}

```

Agora que criamos o componente, chegou o momento de utilizá-lo. Neste caso, criaremos uma página e, em seu método construtor, criaremos um objeto accordion (**TAccordion**). Este objeto accordion pode conter vários outros containers em seu interior. Para adicionar conteúdo, utilizamos o método **appendPage()**, que recebe o título do conteúdo, e o conteúdo em si (**\$page1, \$page2..**).

Neste exemplo, acrescentamos uma tabela na primeira posição; um painel na segunda; e outra tabela na terceira posição. Após adicionar o conteúdo ao accordion, basta acrescentarmos conteúdo a estas tabelas e painéis. Na primeira posição (**\$page1**), que é uma tabela, são adicionados vários campos (**\$field1, \$field2, \$field3**, etc.). Em seguida, estes vários campos são acrescentados à tabela. Por fim, o accordion é adicionado à página.

app/control/Presentation/Extending/AccordionView.class.php

```

<?php
class AccordionView extends TPage
{
    function __construct()
    {
        parent::__construct();

        // cria o accordion
        $accordion = new TAccordion;

        // cria três containers
        $page1 = new TTable;
        $page2 = new TPanel(370,180);
        $page3 = new TTable;

        // adiciona os containers ao accordion
        $accordion->appendPage('Basic data', $page1);
        $accordion->appendPage('Other data', $page2);
        $accordion->appendPage('Other note', $page3);

        // cria alguns campos
        $field1 = new TEntry('field1');
        $field2 = new TEntry('field2');
        $field3 = new TEntry('field3');
        $field4 = new TEntry('field4');
        $field5 = new TEntry('field5');
        $field6 = new TEntry('field6');
        $field7 = new TEntry('field7');
        $field8 = new TEntry('field8');
        $field9 = new TEntry('field9');
        $field10= new TEntry('field10');

        // adiciona o campo
        $row=$page1->addRow();
        $cell=$row->addCell(new TLabel('<b>Table Layout</b>'));
        $cell->valign = 'top';
        $cell->colspan=2;

        // adiciona o campo
        $row=$page1->addRow();
        $row->addCell(new TLabel('Field1:'));
        $row->addCell($field1);

        // adiciona o campo
        $row=$page1->addRow();
        $row->addCell(new TLabel('Field2:'));
        $cell = $row->addCell($field2);
        $cell->colspan=3;

        // adiciona o campo
        $row=$page1->addRow();
        $row->addCell(new TLabel('Field3:'));
        $cell = $row->addCell($field3);
        $cell->colspan=3;

        ...
        // adiciona o accordion na página
        parent::__add($accordion);
    }
}

```

Na figura a seguir, podemos visualizar o resultado do Accordion.

The screenshot shows a user interface element titled 'Basic data'. Below it is a section titled 'Table Layout' containing five input fields labeled 'Field1' through 'Field5'. At the bottom of the main panel, there are two collapsed sections: 'Other data' and 'Other note'. A watermark 'MARCOS ANTONIO RAFAEL DA FONSECA' is visible across the entire interface.

Figura 96 Componente Accordion

4.10 Relatórios

Nesta seção abordaremos a geração de relatórios. Num primeiro momento, veremos como se dá a geração de relatórios em formato de tabelas, e em seguida a conversão de Templates HTML em PDF.

4.10.1 Relatórios tabulares

A utilização de datagrids oferece um meio bastante utilizado para listagem e visualização das informações já registradas pelo sistema. Entretanto, as datagrids não substituem relatórios. Os usuários frequentemente desejam imprimir documentos contendo quantidades maiores de dados a partir do sistema. Neste caso, precisamos usar formatos mais adequados para a impressão. O Adianti Framework disponibiliza um conjunto de classes para geração de relatórios em tabelas nos formatos HTML, PDF, XLS e RTF. O ponto positivo na utilização desta biblioteca é que você escreve o relatório somente uma vez, e pode exportar para os três formatos.

Obs: As classes para geração de relatórios vistas aqui foram propostas pela primeira vez no livro “Criando relatórios com PHP”, do mesmo autor.

Para demonstrar a utilização das classes para geração de relatórios, construiremos um formulário para seleção de clientes. Este formulário permite filtrar por nome, cidade e categoria. O usuário poderá preencher um ou mais filtros neste formulário. O programa deverá buscar todos os clientes na base de dados pelos filtros selecionados. Para tal, utilizaremos o recurso de coleções e critérios, como visto no capítulo 3. A partir da coleção de clientes carregada em memória, procederemos com a escrita (gravação) do relatório. O relatório poderá ser extraído em um dos formatos a seguir: HTML, PDF, RTF ou XLS conforme pode ser visto na figura a seguir.

Relatório tabular
MARCOS ANTONIO RAFAEL DA FONSECA -

Customer

City

Category

Output HTML PDF RTF XLS

MARCOS ANTONIO RAFAEL DA FONSECA -

Figura 97 Formulário de geração de relatório tabular

O programa inicia com a criação do formulário (`BootstrapFormBuilder`). Logo em seguida, são criados os campos do formulário (cliente, cidade, categoria, e formato de saída), e adicionadas algumas opções (`HTML`, `PDF`, `RTF`, `XLS`) no radiobutton de formato, bem como definidas algumas características dos campos como seus tamanhos. O campo de formato virá com `PDF` pré-escolhido, o que se dá pelo método `setValue()`. O método `setMinLength()` define a quantidade mínima de caracteres para a busca, e o método `setUseButton()` define que o radiobutton será exibido como botão.

Após criarmos os objetos, adicionamos os vários campos de filtragem ao formulário por meio do método `addFields()`. Ao final do método construtor, uma ação é criada, vinculada ao método `onGenerate()`, que gerará o relatório.

app/control/Presentation/Report/TabularReportView.class.php

```
<?php
class TabularReportView extends TPage
{
    private $form; // form
    function __construct()
    {
        parent::__construct();
        // cria o formulário
        $this->form = new BootstrapFormBuilder('form_Customer_report');
        $this->form->setFormTitle( _t('Tabular report') );

        // cria os campos de filtro
        $name      = new TEntry('name');
        $city_id   = new TDBUniqueSearch('city_id', 'samples', 'City', 'id', 'name');
        $category_id = new TDBCombo('category_id', 'samples', 'Category', 'id', 'name');
        $output_type = new TRadioGroup('output_type');

        // adiciona os campos ao formulário
        $this->form->addFields( [new TLabel('Customer')],      [$name] );
        $this->form->addFields( [new TLabel('City')],       [$city_id] );
        $this->form->addFields( [new TLabel('Category')],  [$category_id] );
        $this->form->addFields( [new TLabel('Output')],     [$output_type] );

        // define propriedades dos campos
        $name->setSize( '80%' );
        $city_id->setSize( '80%' );
        $category_id->setSize( '80%' );
        $output_type->setUseButton();
        $city_id->setMinLength(1);
        $options = ['html' =>'HTML', 'pdf' =>'PDF', 'rtf' =>'RTF', 'xls' =>'XLS'];
    }
}
```

```
$output_type->addItems($options);
$output_type->setValue('pdf');
$output_type->setLayout('horizontal');

// adiciona a ação ao formulário
$this->form->addAction( 'Generate', new TAction(array($this, 'onGenerate')),
'fa:download blue');

parent::add($this->form);
}
```

O método `onGenerate()` será acionado pelo botão “Gerar” do formulário e será responsável por ler os filtros informados pelo usuário, carregar os objetos `Customer` conforme os filtros e criar o relatório no formato desejado (HTML, PDF, RTF, XLS).

Este método inicialmente abre uma transação com a base de dados (`TTransaction`) e obtém os dados do formulário pelo método `getData()`. Em seguida, é criado um repositório para o Active Record `Customer` e são verificados quais campos do formulário estão preenchidos por meio de uma série de comandos `IF`. A cada campo preenchido, é adicionado um filtro ao critério de seleção de objetos. Após, é utilizado o método `load()` para carregar em memória a coleção de objetos.

Caso algum registro tenha sido encontrado, é verificado o tipo de relatório que o usuário quer gerar por meio de um `switch/case`. Para cada formato de saída escolhido existe uma classe correspondente para geração, que são: `TTableWriterHTML`, `TTableWriterPDF`, `TTableWriterRTF`, e `TTableWriterXLS`. As classes possuem os mesmos métodos e a mesma forma de funcionamento. A diferença está no momento de instanciar os objetos. Cada uma gerará um arquivo em um formato diferente.

A escrita de relatórios tabulares é baseada em linhas, com o método `addRow()` e células, com o método `addCell()`. Sempre que adicionarmos uma célula ao relatório, devemos informar qual o estilo de escrita utilizado. Um estilo identifica fonte, tamanho, formatação e cores. Após instanciar uma das classes `TTableWriter`, partimos para a criação de estilos. Cada estilo será utilizado em uma parte do relatório. Um estilo é criado pelo método `addStyle()`, e recebe como parâmetros: nome do estilo; fonte; tamanho; estilo (negrito, itálico); cor da fonte e cor do fundo.

Após criarmos os estilos, utilizamos o método `setHeaderCallback()` para definir um método de escrita do cabeçalho do relatório, e `setFooterCallback()` para definir um método de escrita do rodapé do relatório. Em casos como PDF e RTF, cabeçalhos e rodapés se repetem no início e fim de todas as páginas. Ambos métodos definem uma função anônima, que recebe um objeto `$table`, e sobre este, cria uma linha com conteúdo, para cabeçalho ou para rodapé, conforme o método executado. Esta função anônima é executada internamente sempre que houver necessidade de iniciar uma nova página, que no caso de HTML e XLS ocorre somente uma vez, mas em PDF e RTF (que possuem o conceito de páginas), pode ocorrer múltiplas vezes.

Para adicionar linhas, basta utilizar o método `addRow()` e para adicionar células dentro das linhas, basta utilizar o método `addCell()`. O método `addCell()` recebe como parâmetros: o conteúdo da célula; o alinhamento; o nome do estilo utilizado, e a quantidade de células a mesclar.

Após definirmos os métodos de cabeçalho e rodapé, percorremos os objetos `$customers` (vetor de Active Record) e para cada um, acrescentamos uma linha com seus atributos. Ao fim, utilizamos o método `save()` para escrever o arquivo em disco e o método `openFile()` da classe `TPage` para apresentar o relatório para o usuário.

```

function onGenerate()
{
    try {
        TTransaction::open('samples');

        // obtém os dados preenchidos no formulário
        $data = $this->form->getData();

        // define os filtros do relatório, baseado no preenchimento do formulário
        $repository = new TRepository('Customer');
        $criteria = new TCriteria;
        if ($data->name) {
            $criteria->add(new TFilter('name', 'like', "%{$data->name}%"));
        }

        if ($data->city_id) {
            $criteria->add(new TFilter('city_id', '=', "{$data->city_id}"));
        }

        if ($data->category_id) {
            $criteria->add(new TFilter('category_id', '=', "{$data->category_id}"));
        }

        // carrega os clientes da base de dados
        $customers = $repository->load($criteria);
        $format = $data->output_type;

        if ($customers)
        {
            $widths = array(40, 200, 80, 120, 80);

            // instancia a classe correta de escrita do relatório
            switch ($format)
            {
                case 'html':
                    $table = new TTableWriterHTML($widths);
                    break;
                case 'pdf':
                    $table = new TTableWriterPDF($widths);
                    break;
                case 'rtf':
                    $table = new TTableWriterRTF($widths);
                    break;
                case 'xls':
                    $table = new TTableWriterXLS($widths);
                    break;
            }

            if (!empty($table))
            {

```


A seguir, você confere parte do relatório gerado no formato PDF.

Customers				
Code	Name	Category	Email	Birthdate
1	Andrei Zmievski	Casual	contact@gmail.com	1980-01-01
2	Rubens Prates	Frequente	contact@gmail.com	1990-01-01
3	Augusto Campos	Frequente	contact@gmail.com	1990-01-01
4	Marcelio Leal	Frequente	contact@gmail.com	1990-01-01
5	Manuel Lemos	Frequente	contact@gmail.com	1990-01-01
6	Fábio Locatelli	Frequente	contact@gmail.com	1990-01-01
7	Leonardo Soldatelli	Frequente	contact@gmail.com	1990-01-01
8	Alberto Bengoa	Frequente	contact@gmail.com	1990-01-01
9	Fábio Milani	Frequente	contact@gmail.com	1990-01-01
10	Huberto Meyer	Frequente	contact@gmail.com	1990-01-01

Figura 98 Relatório gerado no formato PDF

4.10.2 Relatório sobre consulta SQL

Uma situação em que o usos e instruções SQL é praticamente indispensável é em relatórios. Por que em relatórios frequentemente precisamos cruzar muitas tabelas, fazer uso de subconsultas, expressões matemáticas, bem como recursos específicos do banco de dados (funções de georreferência, geométricas, data warehouse, etc). Além disso, existem consultas cujo resultado só faz sentido a partir do cruzamento de várias tabelas. Realizar tais operações usando apenas objetos teria um custo muito alto, tendo em vista que para montar um relatório teríamos de percorrer milhares de objetos de uma classe e, a partir desses, buscar informações em objetos relacionados. Claro que essas questões de desempenho caem por terra com o advento de novas tecnologias que permitem o uso de banco de dados em memória. Mas como isso ainda não é uma realidade para as maiorias, vamos adotar uma estratégia simples e funcional.

Neste próximo exemplo, criaremos um relatório tabular sobre uma consulta SQL manual. Esta abordagem é apropriada para casos em que precisamos executar uma lógica mais complexa para trazer os dados do banco de dados. Embora fácil de utilizar, preferimos indicar aos desenvolvedores encapsularem consultas complexas dentro de Views, que será a técnica abordada na seção seguinte.

O programa que gerará o relatório manual iniciará com um formulário com um filtro por cidade. O código da cidade selecionada será utilizado como filtro da consulta manual. Teremos ainda neste formulário um campo para seleção do formato de saída. Para a seleção da cidade, utilizaremos o componente `TDBUniqueSearch` para realizar a busca de registro, trazendo o ID selecionado da cidade. Já para o formato de saída, utilizaremos um campo de rádio (`TRadioGroup`). O formulário terá um botão de ação para gerar o relatório, vinculado ao método `onGenerate()`.

app/control/Presentation/Report/TabularReportView.class.php

```
<?php
class TabularReportQueryView extends TPage
{
    private $form; // form

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

        // cria o formulário
        $this->form = new BootstrapFormBuilder('form_Customer_report');
        $this->form->setFormTitle( _t('Tabular report') );

        // cria os campos do form
        $city_id      = new TDBUniqueSearch('city_id', 'samples', 'City', 'id', 'name');
        $output_type  = new TRadioGroup('output_type');

        $this->form->addFields( [new TLabel('City')],      [$city_id] );
        $this->form->addFields( [new TLabel('Output')],     [$output_type] );

        // define propriedades dos campos
        $city_id->setSize( '80%' );
        $city_id->setMinLength(0);
        $output_type->setUseButton();
        $options = ['html' =>'HTML', 'pdf' =>'PDF', 'rtf' =>'RTF', 'xls' =>'XLS'];
        $output_type->addItems($options);
        $output_type->setValue('pdf');
        $output_type->setLayout('horizontal');
        $city_id->addValidation( 'City', new TRequiredValidator );
        $this->form->addAction( 'Generate', new TAction(array($this, 'onGenerate')), 'fa:download blue' );

        parent::add($this->form);
    }
}
```

O método `onGenerate()`, por sua vez obterá os dados preenchidos no formulário pelo método `getData()`. A variável `$data`, terá o código da cidade, e o formato de saída escolhido. Abrimos uma transação com o método `TTransaction::open()`, e armazenamos a variável de conexão `$source`. Em seguida, definimos a consulta SQL que será realizada no banco de dados. Perceba que a variável `:city_id` será injetada posteriormente durante a execução da query, sendo substituída com o conteúdo da cidade escolhida pelo formulário. O método `TDatabase::getData()` obtém os dados da consulta. Para tal, ele recebe o conector (`$source`), a consulta (`$query`), um parâmetro opcional com mapeamento de campos entre os aliases da consulta e o vetor resultante (`null` para default), e um vetor com os parâmetros a serem injetados na consulta (`(city_id)`). Veja que aqui, estamos injetando somente o código da cidade. Esta operação resultará em uma matriz com os dados da consulta (`$rows`).

Todo o código em seguida já foi abordado anteriormente quando geramos um relatório sobre uma classe de modelo. Basicamente, fazemos um switch/case sobre o formato escolhido, para instanciar a classe apropriada. Em seguida, são adicionados estilos de formatação, que serão usados no método `addCell()` para definir a formatação das células.

Os métodos `setHeaderCallback()` e `setFooterCallback()` são utilizados para definir um método a ser executado para gerar cabeçalhos e rodapés, sempre que uma página iniciar (cabeçalho) ou encerrar (rodapé).

Por fim, os dados são percorridos por um foreach, e adicionados à tabela com os métodos `addRow()` para adicionar linha, e `addCell()` para adicionar célula. A gravação é feita pelo método `save()`, que gera o arquivo de saída. O método `openFile()` é usado para realizar a abertura do arquivo.

```

function onGenerate()
{
    try
    {
        // obtém os dados do formulário
        $data = $this->form->getData();
        $this->form->setData($data);

        $format = $data->output_type;

        // abre transação com a base
        $source = TTransaction::open('samples');

        // define a consulta SQL
        $query = "SELECT cs.id as 'id',
                    cs.name as 'name',
                    cs.email as 'email',
                    cs.birthdate as 'birthdate',
                    ct.name as 'category_name'
            FROM customer cs, category ct
            WHERE cs.category_id = ct.id and cs.city_id = :city_id";

        $rows = TDatabase::getData($source, $query, null,
                                   [ 'city_id' => $data->city_id ]);

        if ($rows)
        {
            $widths = array(40, 200, 80, 120, 80);

            switch ($format)
            {
                case 'html':
                    $table = new TTableWriterHTML($widths);
                    break;
                case 'pdf':
                    $table = new TTableWriterPDF($widths);
                    break;
                case 'rtf':
                    $table = new TTableWriterRTF($widths);
                    break;
                case 'xls':
                    $table = new TTableWriterXLS($widths);
                    break;
            }

            if (!empty($table))
            {
                // cria os estilos
                $table->addStyle('header', 'Helvetica', '16', 'B', ...);
                $table->addStyle('title', 'Helvetica', '10', 'B', ...);

```

```

// cabeçalhos e rodapés
$table->setHeaderCallback( function($table) {
    $table->addRow();
    $table->addCell('Customers', 'center', 'header', 5);

}); ...

$table->setFooterCallback( function($table) {
    $table->addRow();
    $table->addCell(date('Y-m-d h:i:s'), 'center', 'footer', 5);
});

// controls the background filling
$colour= FALSE;

// percorre as linhas de dados
foreach ($rows as $row)
{
    $style = $colour ? 'datap' : 'data1';
    $table->addRow();
    $table->addCell($row['id'], 'center', $style);
    $table->addCell($row['name'], 'left', $style);
    $table->addCell($row['category_name'], 'center', $style);
    $table->addCell($row['email'], 'left', $style);
    $table->addCell($row['birthdate'], 'center', $style);

    $colour = !$colour;
}

$output = "app/output/tabular.{$format}";

// grava o arquivo
if (!file_exists($output) OR is_writable($output))
{
    $table->save($output);
    parent::openFile($output);
}
else
{
    throw new Exception(_t('Permission denied') . ' : ' . $output);
}

// exibe diálogo de sucesso
new TMessage('info', "Report generated");
}

}

else
{
    new TMessage('error', 'No records found');
}

// fecha transação
TTransaction::close();
}

catch (Exception $e)
{
    new TMessage('error', $e->getMessage());
    TTransaction::rollback();
}
}
}

```

4.10.3 Relatório sobre View

Os exemplos anteriores mostraram como criar um relatório sobre uma classe Active Record, e também sobre uma consulta SQL manual. Precisamos recorrer ao SQL quando temos uma consulta com uma lógica mais complexa que envolva junção de diferentes tabelas, subconsultas, dentre outros elementos que tornariam a aplicação complexa demais. Porém, não recomendamos aos desenvolvedores escreverem o SQL diretamente no código-fonte. Uma alternativa bastante interessante para construir relatórios sobre consultas SQL é utilizando Views. Uma vantagem das Views, é a facilidade de organização, visto que é possível alterar regras de carregamento de dados e cálculos, sem necessidade de alterar a programação. Além disso, elas tornam o código-fonte mais limpo, visto que não teremos a necessidade de ter SQL espalhado no código-fonte da aplicação.

O primeiro passo para criar o relatório, é criar a View.

```
CREATE VIEW view_sales AS
SELECT id,          name,
       address,    phone,
       birthdate,   status,
       email,       gender,
       city_id,     category_id,
       (SELECT sum(total)
        FROM sale
        WHERE customer_id = customer.id ) AS total,
       (SELECT max(date)
        FROM sale
        WHERE customer_id = customer.id ) AS last_date
FROM customer
```

Em seguida, precisamos definir uma classe de modelo (Active Record) sobre a View.

app/model/ViewSales.php

```
<?php
class ViewSales extends TRecord
{
    const TABLENAME = 'view_sales';
    const PRIMARYKEY = 'id';
    const IDPOLICY = 'max'; // {max, serial}

    public function __construct($id = NULL, $scallObjectLoad = TRUE)
    {
        parent::__construct($id, $scallObjectLoad);
        parent::addAttribute('name');
        parent::addAttribute('address');
        parent::addAttribute('phone');
        parent::addAttribute('birthdate');
        parent::addAttribute('status');
        parent::addAttribute('email');
        parent::addAttribute('gender');
        parent::addAttribute('city_id');
        parent::addAttribute('category_id');
        parent::addAttribute('total');
        parent::addAttribute('last_date');
    }
}
```

Por fim, podemos repetir a mesma abordagem utilizada anteriormente ao construirmos relatórios tabulares. No lugar de construir o relatório sobre uma classe que representa uma tabela, agora faremos sobre uma classe que representa uma View.

```
public function onGenerate()
{
    try {
        $repository = new TRepository('ViewSales');
        $criteria  = new TCriteria;

        if ($data->city_id) {
            $criteria->add(new TFilter('city_id', '=', $data->city_id));
        }

        if ($data->category_id) {
            $criteria->add(new TFilter('category_id', '=', $data->category_id));
        }

        $customers = $repository->load($criteria);
        ...
    }
}
```

4.10.4 Conversão de Templates para PDF

No exemplo anterior, vimos como gerar relatórios em formato de tabela em HTML, PDF, RTF, e XLS. O formato tabular é bom para dados lineares e em grande quantidade. Porém, nem sempre o formato tabular é o mais adequado para relatórios e documentos quando precisamos de uma liberdade maior de criação. Às vezes precisamos de um design mais livre do conteúdo em relação ao documento.

Para apresentar uma forma mais livre de criação de relatórios e documentos, neste exemplo vamos utilizar um Template HTML para montar um relatório, e em seguida converter o mesmo para o formato PDF, e apresentá-lo em uma nova janela para que o usuário possa visualizar, salvar ou imprimir este arquivo.

O exemplo inicia com a utilização da classe `THtmlRenderer` para realizar o carregamento de um Template HTML (`customer_accounts.html`). Em seguida, utilizamos o vetor `$replace` para indicar como o Template será preenchido. Como este exemplo já foi explicado anteriormente no exemplo sobre “Template View com matrizes”, vamos focar na conversão do documento em si, não em sua geração.

Após definir pelo vetor `$replace`, como o Template terá suas seções e variáveis preenchidas, utilizamos o método `enableSection()` para habilitar o Template, e preenche-lo por meio da variável `$replace`. O Template é processado por meio do método `getContents()`, que retornará o resultado do processamento já com o conteúdo.

Para gerar o documento no formato PDF, utilizamos a classe `Dompdf`. Esta classe possui o método `loadHtml()` para carregar o conteúdo HTML, `setPaper()` para definir as propriedades da página, `render()` para renderizar o PDF, e `output()` para gerar a saída do documento, que é escrita em disco pelo método `file_put_contents()`.

Por fim, criamos uma janela (`TWindow`), com o documento em seu interior.

`app/control/Presentation/Report/DocumentHtmlPdfView.class.php`

```
<?php
class DocumentHtmlPdfView extends TPage
{
    public function __construct()
    {
        parent::__construct();
        // carrega o Template
        $this->html = new THtmlRenderer('app/resources/customer_accounts.html');

        try {
            // define as substituições do Template
            $plain_object = new stdClass;
            $plain_object->id = '001';
            $plain_object->name = 'John';
            $replace = array();
            $replace['object'] = $plain_object;
            $replace['header'] = [ [ 'name' => 'Field test',
                'value' => 'Field value' ] ];
            $replace['accounts1'] = [ [ 'date'=>'2016-05-01',
                'value'=>100,
                'details' => [ [ 'product'=> 'Chocolate',
                    'qty'=> 10,
                    'value' => 5 ],
                    [ 'product'=> 'Milk',
                    'qty'=> 5,
                    'value' => 10 ] ] ],
            [ 'date' => '2016-05-03',
                'value' => 200,
                'details' => [ [ 'product' => 'Cofee',
                    'qty' => 10,
                    'value' => 10 ],
                    [ 'product'=> 'Pizza',
                    'qty' => 5,
                    'value' => 20 ] ] ] ];
            $this->html->enableSection('main', $replace);
            parent::add($this->html);

            // lê o conteúdo já processado do Template
            $contents = $this->html->getContents();
            // converte o HTML em PDF
            $dompdf = new \Dompdf\Dompdf();
            $dompdf->loadHtml($contents);
            $dompdf->setPaper('A4', 'portrait');
            $dompdf->render();
            file_put_contents('tmp/document.pdf', $dompdf->output());

            // abre uma janela para exibir o PDF
            $window = TWindow::create(_t('Document HTML->PDF'), 0.8, 0.8);
            $object = new TElement('object');
            $object->data = 'tmp/document.pdf';
            $object->type = 'application/pdf';
            $object->style = "width: 100%; height:calc(100% - 10px)";
            $window->add($object);
            $window->show();
        }
        catch (Exception $e) {
            new TMessage('error', $e->getMessage());
        }
    }
}
```

A seguir, podemos conferir o resultado da execução deste programa.

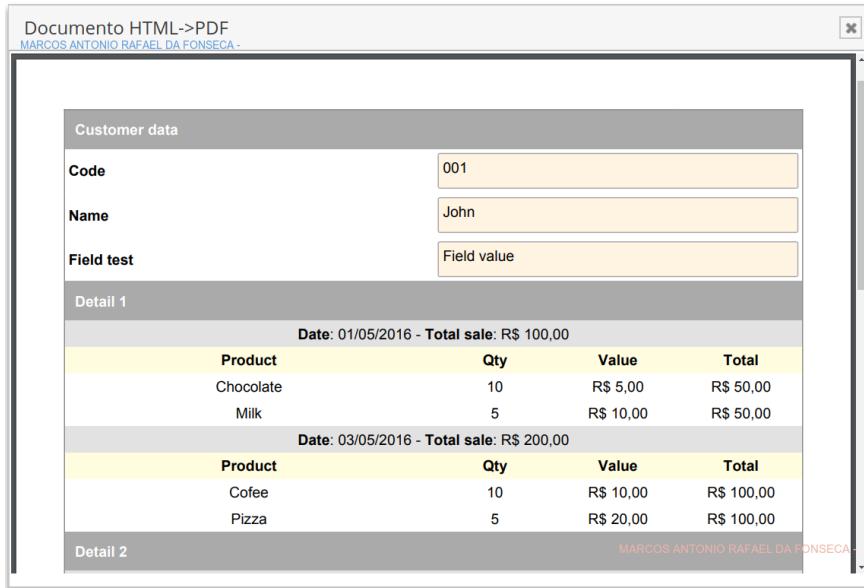


Figura 99 Documento PDF gerado em uma Janela

4.10.5 Fatura em tela e para impressão

Neste exemplo, vamos demonstrar como criar um documento de fatura usando um Template HTML. O objetivo do exemplo será montar um documento em tela, e um botão para converter o documento em PDF. O documento terá em seu cabeçalho os dados da fatura (data, método de pagamento), do cliente (nome, endereço, cidade), e da entrega (endereço, cidade, etc). Abaixo do cabeçalho do documento, teremos os itens da fatura (produtos vendidos), com informações como código, preço e quantidade. Ao final, serão apresentados alguns totalizadores.

Para construir este exemplo, usaremos um Template em HTML (`invoice.html`) utilizando a Biblioteca Bootstrap, o que garantirá um visual agradável, além da responsividade.

Este exemplo começa com a leitura do Template da fatura (`invoice.html`). Em seguida, vamos simular alguns objetos do banco de dados como a fatura (`$invoice`), o cliente (`$customer`) e a entrega (`$shipping`). Estes objetos serão injetados no Template, mas podem ser facilmente substituídos por objetos do banco de dados na prática.

Os itens da fatura são representados pela matriz `$replace`, que contém um vetor com as linhas (itens) da fatura. Cada item, é um outro vetor indexado pelo nome do atributo de cada item. Estes itens poderiam ser facilmente substituídos pelo retorno do banco de dados, usando os métodos já vistos.

app/control/Presentation/Extending/TemplateInvoiceView.class.php

```

<?php
class TemplateInvoiceView extends TPage
{
    public function __construct()
    {
        parent::__construct();

        // Lê o template
        $this->html = new THtmlRenderer('app/resources/invoice.html');

        try
        {
            // Cria objetos para injetar no template
            $invoice = new stdClass;
            $invoice->id = '001002003';
            $invoice->date = '2019-03-20';
            $invoice->order_date = '2019-03-20';
            $invoice->pay_method = 'Paypal';
            $invoice->pay_account = 'john@email.com';
            $invoice->shipping = 25;

            $customer = new stdClass;
            $customer->name = 'John gray';
            $customer->address = 'Nice Street, 123';
            $customer->complement = 'Apt. 456';
            $customer->city = 'Springfield, ST 54321';

            $shipping = new stdClass;
            $shipping->name = 'Jane white';
            $shipping->address = 'Nice Street, 234';
            $shipping->complement = 'Apt. 123';
            $shipping->city = 'Springfield, ST 54321';

            $replace = array();
            $replace['invoice'] = $invoice;
            $replace['customer'] = $customer;
            $replace['shipping'] = $shipping;

            $replace['items'] = [ [ 'code' => '001',
                                  'description' => 'Chocolate',
                                  'price' => 100,
                                  'quantity' => 1 ],
                                 [ 'code' => '002',
                                   'description' => 'Cofee',
                                   'price' => 100,
                                   'quantity' => 2 ],
                                 ... ];

            // substitui a matriz de dados no documento
            $this->html->enableSection('main', $replace);

            $panel = new TPanelGroup('Invoice');
            $panel->addHeaderActionLink('Export', new TAction([$this, 'onExportPDF'],
                                                               ['static' => '1']), 'fa:save' );
            $panel->add($this->html);

            parent::__add($panel);
        }
        catch (Exception $e) {
            new TMessage('error', $e->getMessage());
        }
    }
}

```

Neste exemplo, teremos um botão de ação no cabeçalho do painel, para exportar o conteúdo para PDF. Este botão está vinculado ao método `onExportPDF`. O método `onExportPDF` clona o HTML do Template. Esta cópia será convertida em PDF. Para tal, antes concatenamos o conteúdo do arquivo `styles-print.html`. Este arquivo contém estilos CSS ajustados para impressão. Sem esses estilos, o resultado do PDF não ficará com o layout correto, e você verá um conteúdo desorganizado.

Utilizamos a classe `Dompdf` para converter o conteúdo. Seu método `loadHtml()` carrega o HTML, e o método `render()` renderiza o conteúdo em PDF em memória. Já o método `output()` gera a saída do PDF, que é escrita pelo `file_put_contents()`. Por fim, criamos um objeto janela `TWindow` e adicionamos o arquivo gerado dentro, para fornecer uma visualização agradável.

```
public function onExportPDF($param)
{
    try
    {
        // obtém o HTML final
        $html = clone $this->html;
        $contents = file_get_contents('app/resources/styles-print.html') .
                    $html->getContents();

        // Converte o HTML em PDF
        $dompdf = new \Dompdf\Dompdf();
        $dompdf->loadHtml($contents);
        $dompdf->setPaper('A4', 'portrait');
        $dompdf->render();

        $file = 'app/output/invoice.pdf';

        // escreve em arquivo o PDF
        file_put_contents($file, $dompdf->output());

        // abre o PDF em janela
        $window = TWindow::create('Invoice', 0.8, 0.8);
        $object = new TElement('object');
        $object->data = $file;
        $object->type = 'application/pdf';
        $object->style = "width: 100%; height:calc(100% - 10px)";
        $window->add($object);
        $window->show();
    }
    catch (Exception $e)
    {
        new TMessage('error', $e->getMessage());
    }
}
```

O Template HTML utilizado para a montagem da fatura possui uma sessão inicial com dados de cabeçalho. Neste exemplo, vamos cortar boa parte e exibir somente o trecho com os dados do cliente, pois as demais partes são similares. Neste exemplo, podemos ver os dados do ID da fatura, data, e dados do cliente.

app/resources/invoice.html

```
<!--[main]-->
...
<div class="invoice-title text-center mb-3">
    <h2>Invoice #{{invoice->id}}</h2>
    <strong>Date:</strong> date_format({{invoice->date}}, 'd/m/Y')
```

```
</div>
<hr>
<div class="row">
    <div class="col-sm-6">
        <h3>Billed To:</h3>
        <address>
            {{customer->name}}<br>
            {{customer->address}}<br>
            {{customer->complement}}<br>
            {{customer->city}}<br>
        </address>
    </div>
</div>
...

```

Em seguida, temos os dados dos itens da fatura. Neste ponto, temos uma tabela, na qual o body da tabela terá os itens repetidos (código, descrição, preço, quantidade). A sessão **[items]** será repetida pois é alimentada por uma matriz (**\$replace['items']**), que por sua vez tem muitas linhas de dados. Logo, esta seção será repetida a quantidade de vezes conforme a quantidade de itens da matriz. Aqui, utilizamos ainda o comando **%set%** para produzir um subtotal, resultante de um cálculo do preço e da quantidade.

Item	Description	Price	Quantity	Totals
!---[items]-->				
!---[/items]-->				

```
<table class="table table-condensed" style="border-collapse:collapse">
    <thead>
        <tr>
            <td><strong>Item</strong></td>
            <td><strong>Description</strong></td>
            <td class="text-center"><strong>Price</strong></td>
            <td class="text-center"><strong>Quantity</strong></td>
            <td class="text-right"><strong>Totals</strong></td>
        </tr>
    </thead>
    <tbody>
        <!--[items]-->
        <tr>
            <td>{{code}}</td>
            <td>{{description}}</td>
            <td class="text-center">{{price}}</td>
            <td class="text-center">{{quantity}}</td>
            <td class="text-right">
                R$ number_format(evaluate( {{price}} * {{quantity}} ), 2, ',', '.')
            </td>
            <!-- {% set subtotal += evaluate( {{price}} * {{quantity}} ) %} -->
        </tr>
        <!--[/items]-->
    </tbody>
</table>
```

A figura a seguir demonstra a fatura gerada em tela.

Billed To:
John gray
Nice Street, 123
Apt. 456
Springfield, ST 54321

Shipped To:
Jane white
Nice Street, 234
Apt. 123
Springfield, ST 54321

Payment Method:
Paypal
john@email.com

Order Date:
20/03/2019

Order summary				
Item	Description	Price	Quantity	Totals
001	Chocolate	100	1	R\$ 100,00
002	Coffee	100	2	R\$ 200,00
003	Water	100	3	R\$ 300,00
			Subtotal	R\$ 600,00
			Shipping	R\$ 25,00
			Total	R\$ 625,00

MARCOS ANTONIO RAFAEL DA FONSECA -

Figura 100 Fatura

4.11 Gráficos

Nos próximos exemplos, será demonstrado como gerar gráficos pelo Framework. Serão três tipos: linhas, barras e pizza. Para os três tipos de gráficos, serão utilizados Templates HTML que por sua vez utilizam a API do Google para geração de gráficos. Ao final, montaremos um painel com vários gráficos em uma única tela.

4.11.1 Gráfico de linhas

Para gerar gráficos de linhas, utilizaremos a API do Google. As chamadas necessárias encontram-se no arquivo `google_line_chart.html`. Somente precisaremos formatar devidamente os dados e passar para este Template, que já contém algumas variáveis como `[$data]` para receber os dados, `[$title]` para receber o título, e `[$width]` e `[$height]` para largura e altura, por exemplo.

O conteúdo mais relevante é definido pela matriz `$data`, que contém os dados, bem como as legendas do gráfico. Na primeira linha, contém o título do eixo X na primeira posição (Day) e os demais títulos das séries numéricas (Value 1, Value 2, Value 3). Nas demais linhas, a primeira posição representa o nome do ponto no eixo X, seguido dos valores a serem plotados no gráfico para cada série numérica.

O vetor `$data` precisa ser transformado no formato JSON pela função `json_encode()`, visto que é neste formato que a API espera receber os dados.

app/control/Presentation/Chart/LineChartView.class.php

```
<?php
class LineChartView extends TPage
{
    function __construct()
    {
        parent::__construct();

        // instancia o renderizador
        $html = new THtmlRenderer('app/resources/google_line_chart.html');

        // define matriz com dados e legendas
        $data = array();
        $data[] = [ 'Day', 'Value 1', 'Value 2', 'Value 3' ];
        $data[] = [ 'Day 1', 120, 140, 160 ];
        $data[] = [ 'Day 2', 100, 120, 140 ];
        $data[] = [ 'Day 3', 140, 160, 180 ];

        // substitui as variáveis no template
        $html->enableSection('main', array('data' => json_encode($data),
            'width' => '100%',
            'height' => '300px',
            'title' => 'Accesses by day',
            'ytitle' => 'Accesses',
            'xtitle' => 'Day',
            'uniqid' => uniqid()));

        // adiciona o painel na página
        parent::add($html);
    }
}
```

Na figura a seguir, podemos ver o gráfico gerado.

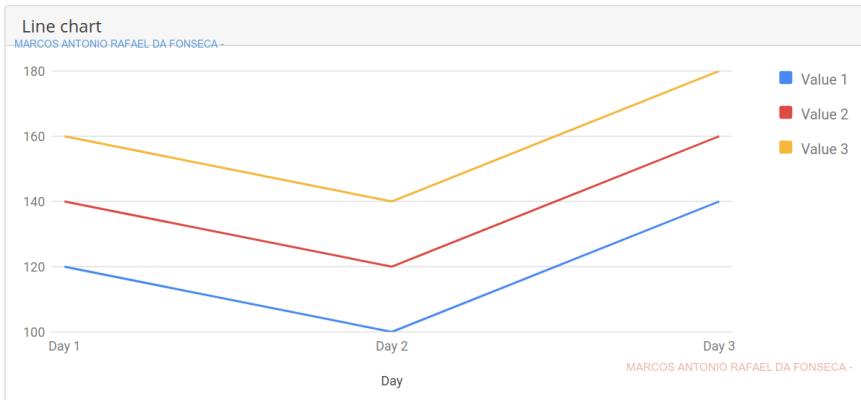


Figura 101 Gráfico de linhas

4.11.2 Gráfico de barras

Para gerar gráficos de linhas, utilizaremos a API do Google. As chamadas necessárias encontram-se no arquivo `google_bar_chart.html`. Somente precisaremos formatar devidamente os dados e passar para este Template, que já contém algumas variáveis como `[$data]` para receber os dados, `[$title]` para receber o título, e `[$width]` e `[$height]` para largura e altura, por exemplo.

Da mesma maneira que no exemplo anterior, a variável `$data` contém os dados, bem como as legendas do gráfico. Na primeira linha, contém o título do eixo X na primeira posição, (Day) e os demais títulos das séries numéricas (Value 1, Value 2, Value 3). Nas demais linhas, a primeira posição representa o nome do ponto no eixo X, seguido dos valores a serem plotados no gráfico para cada série numérica.

O vetor `$data` precisa ser transformado no formato JSON pela função `json_encode()`, visto que é neste formato que a API espera receber os dados.

```
app/control/Presentation/Chart/BarChartView.class.php
-----
<?php
class BarChartView extends TPage
{
    function __construct()
    {
        parent::__construct();

        // instancia o renderizador
        $html = new THtmlRenderer('app/resources/google_bar_chart.html');

        // define a matriz de dados e legenda
        $data = array();
        $data[] = [ 'Day', 'Value 1', 'Value 2', 'Value 3' ];
        $data[] = [ 'Day 1', 100, 120, 140 ];
        $data[] = [ 'Day 2', 120, 140, 160 ];
        $data[] = [ 'Day 3', 140, 160, 180 ];

        // substitui as variáveis no template
        $html->enableSection('main', array(
            'data' => json_encode($data),
            'width' => '100%',
            'height' => '300px',
            'title' => 'Accesses by day',
            'ytitle' => 'Accesses',
            'xtitle' => 'Day',
            'uniqid' => uniqid()));
    }

    // adiciona o painel na página
    parent::__add($html);
}
}
```

Na figura a seguir, podemos ver o gráfico gerado.

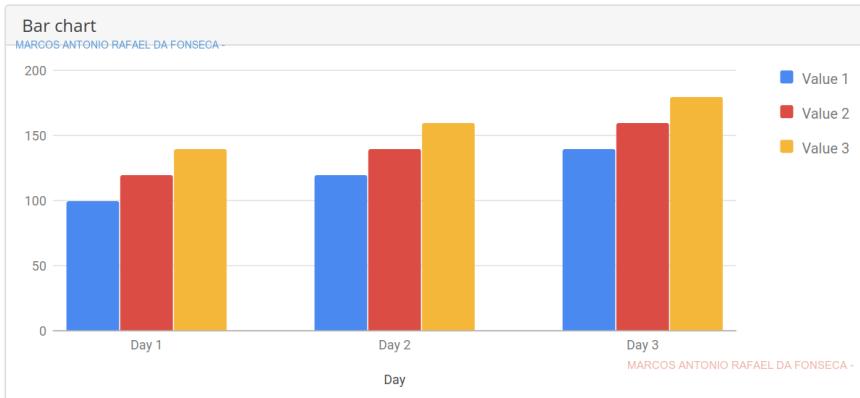


Figura 102 Gráfico de barras

4.11.3 Gráfico de pizza

Para gerar gráficos de linhas, utilizaremos a API do Google. As chamadas necessárias encontram-se no arquivo `google_pie_chart.html`. Somente precisaremos formatar devidamente os dados e passar para este template, que já contém algumas variáveis como `[$data]` para receber os dados, `[$title]` para receber o título, e `[$width]` e `[$height]` para largura e altura, por exemplo.

Diferentemente dos gráficos anteriores, a geração do gráfico de pizza é mais simples. Somente precisaremos criar uma matriz, onde cada posição corresponde a uma fatia do gráfico. Cada uma das posições contém um novo Array no qual a primeira posição representa o nome da fatia do gráfico, e a segunda posição representa o valor que determinará também o tamanho da fatia.

app/control/Presentation/Chart/PieChartView.class.php

```
<?php
class PieChartView extends TPage
{
    function __construct()
    {
        parent::__construct();

        // instancia o renderizador
        $html = new THtmlRenderer('app/resources/google_pie_chart.html');

        // define a matriz de dados e legenda
        $data = array();
        $data[] = [ 'Pessoa', 'Value' ];
        $data[] = [ 'Pedro', 40 ];
        $data[] = [ 'Maria', 30 ];
        $data[] = [ 'João', 30 ];
    }
}
```

```

// substitui as variáveis no template
$html->enableSection('main', array(
    'data' => json_encode($data),
    'width' => '100%',
    'height' => '300px',
    'title' => 'Accesses by day',
    'ytitle' => 'Accesses',
    'xtitle' => 'Day',
    'uniqid' => uniqid()));
}

}

// Adiciona o painel na página
parent::add($html);
}
}

```

Na figura a seguir, podemos ver o gráfico gerado.

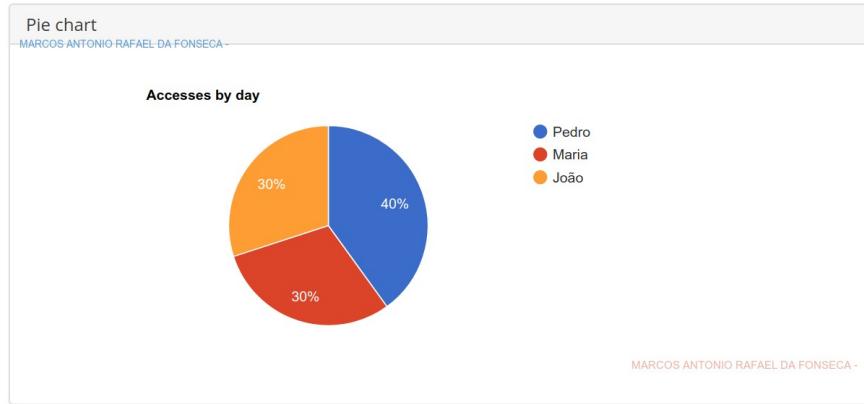


Figura 103 Gráfico de pizza

4.11.4 Dashboard

Nos exemplos anteriores, vimos como construir gráficos individuais. No entanto, é cada vez mais comum a necessidade de apresentar informações sob diferentes formas de visualização, e reunidas na mesma tela na forma de um dashboard (painele). Para criar um painel, vamos reunir os gráficos já construídos, o que será bastante simples.

Ao sabermos o nome da classe de controle de cada gráfico individual, podemos instanciá-los e reunir estes gráficos em uma mesma página. Neste exemplo, vamos instanciar os gráficos individualmente, o que retornará a página contendo aquele gráfico. Com base nestes retornos, podemos adicionar estes objetos em outros containers. Neste caso, vamos adicionar em um objeto do tipo `div`.

Assim, instanciamos individualmente objetos das classes já criadas anteriormente: `BarChartView`, `LineChartView`, e `PieChartView`, passando `FALSE` como parâmetro do método construtor, parâmetro que indicará para cada classe não exibir o caminho de migalha de pão (breadcrumb) no topo. Ao final, adicionamos o conteiner à página.

app/control/Presentation/Chart/DashboardView.class.php

```
<?php
class DashboardView extends TPage
{
    function __construct()
    {
        parent::__construct();

        $vbox = new TVBox;
        $vbox->style = 'width: 100%';

        $div = new TElement('div');
        $div->class = "row";

        $div->add( $g1 = new BarChartView(false) );
        $div->add( $g2 = new LineChartView(false) );
        $div->add( $g3 = new ColumnChartView(false) );
        $div->add( $g4 = new PieChartView(false) );

        $g1->class = 'col-sm-6';
        $g2->class = 'col-sm-6';
        $g3->class = 'col-sm-6';
        $g4->class = 'col-sm-6';

        $vbox->add($div);
        parent::__add($vbox);
    }
}
```

Na figura a seguir, podemos visualizar o resultado deste programa.

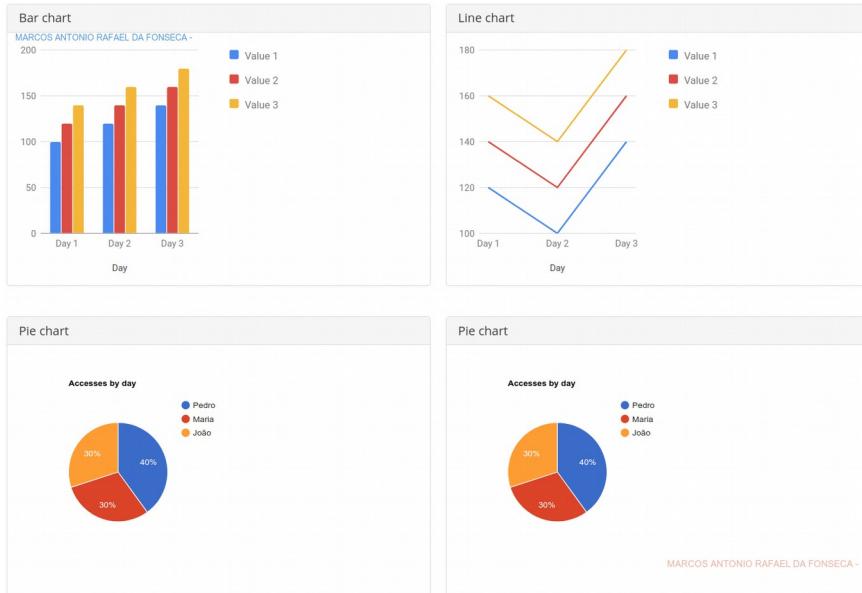


Figura 104 Dashboard

4.12 Etiquetas

Nesta seção, demonstraremos como gerar etiquetas para impressão de códigos de barras e QRCode por meio da classe `AdiantiBarcodeDocumentGenerator`, uma classe que gera tanto Barcodes, quanto QRCodes, baseada em modelos.

4.12.1 Etiquetas de Códigos de barras

Para demonstrar a criação de etiquetas de códigos de barras, vamos utilizar o cadastro já existente de produtos, e apresentar um código de barras baseado em seu ID. A classe `AdiantiBarcodeDocumentGenerator`, responsável pela geração dos códigos de barras, aceita diferentes padrões de barras, tais como: 'C39', 'C39+', 'C39E', 'C39E+', 'C93', 'S25', 'S25+', 'I25', 'I25+', 'C128', 'C128A', 'C128B', 'C128C', 'EAN2', 'EAN5', 'EAN8', 'EAN13', 'UPCA', 'UPCE', 'MSI', 'MSI+', 'POSTNET', 'PLANET', 'RMS4CC', 'KIX', 'IMB', 'CODABAR', 'CODE11', 'PHARMA', 'PHARMA2T'. Além de diferentes padrões, a classe também permite que configuremos um modelo (template) para geração das etiquetas. Cada etiqueta pode conter outras informações além das barras, tais como códigos, descrições e outros.

Para criar as etiquetas de códigos de barras, vamos antes criar um formulário, onde o usuário poderá configurar o método (padrão) de geração dos códigos de barras, e um modelo (template), onde indicará quais informações deseja exibir nas etiquetas. A figura a seguir demonstra este formulário, com dois campos. No campo do template, ele poderá inserir campos como `${id}`, mas também a posição onde entrará o código de barras com a marcação `#barcode#`.

The screenshot shows a Bootstrap-based form for generating barcodes. At the top, there's a label "Barcode" and a note "MARCOS ANTONIO RAFAEL DA FONSECA -". Below this, there are two main input fields. The first field is labeled "Method" and contains the value "EAN13". The second field is labeled "Template" and contains the following code:

```
<b>Código</b>: ${id}
<b>Nome</b>: ${description}
#barcode#
${barcode}
```

At the bottom left of the form is a button labeled "Send", and at the bottom right is the same note: "MARCOS ANTONIO RAFAEL DA FONSECA -".

Figura 105 Formulário para geração de códigos de barras

O código fonte inicia com a montagem do formulário (`BootstrapFormBuilder`). Em seguida, dois campos são criados, para a definição do método de geração, e do template. A variável `$label` contém uma sugestão inicial de Template de geração, com alguns campos que temos a certeza que existem na tabela de produtos.

O formulário terá uma ação vinculada ao método `onSend()`, que gerará um PDF com os códigos de barras e apresentará este PDF em uma janela.

app/control/Presentation/Label/FormBarcodeView.class.php

```
<?php

class FormBarcodeView extends TPage
{
    private $form;
    function __construct()
    {
        parent::__construct();
        $this->form = new BootstrapFormBuilder;
        $this->form->setFormTitle(_t('Barcode'));

        // criação dos campos
        $method = new TCombo('method');
        $template = new TText('template');
        // métodos de geração de código de barras
        $method->addItems( [ 'C39' => 'C39', 'C39+' => 'C39+', 'C39E' => 'C39E', 'C39E+' => 'C39E+', 'C93' => 'C93', 'S25' => 'S25', 'S25+' => 'S25+', 'I25' => 'I25'... ] );

        // adiciona os campos ao formulário
        $this->form->addField([new TLabel('Method')], [$method]);
        $this->form->addField([new TLabel('Template')], [$template]);

        $template->setSize('100%', 100);

        $label = '' . "\n";
        $label .= '<b>Código</b>: {$id}' . "\n";
        $label .= '<b>Nome</b>: {$description}' . "\n";
        $label .= '#barcode#' . "\n";
        $label .= ' ' . "{$barcode}";

        $method->setValue('EAN13');
        $template->setValue($label);

        // define a ação do formulário
        $this->form->addAction('Send', new TAction(array($this, 'onSend')), 'fa:check-circle-o green');

        parent::add($this->form);
    }
}
```

O método `onSend()` será executado a partir do botão de ação do formulário e gerará os códigos de barras. Este método inicia obtendo os dados do formulário, pelo método `getData()`, e em seguida define um vetor de propriedades da etiqueta. Dentre algumas propriedades estão: as margens da etiqueta, sua largura e altura, espaço entre etiquetas, e quantas linhas e colunas de etiquetas serão apresentadas por página.

A classe `AdiantiBarcodeDocumentGenerator` faz o processo de geração. Para tal, ela recebe as propriedades pelo método `setProperties()`, e o modelo de geração da etiqueta pelo método `setLabelTemplate()`. Por último, e mais importante, é necessário passar para a classe, cada um dos objetos que gerarão etiquetas, pois cada objeto gerará uma etiqueta correspondente. O método `addObject()` adiciona um objeto ao gerador de etiquetas. Os objetos neste caso são todos os produtos, que foram lidos pelo método `Product::all()`. Por fim, o método `setBarcodeContent()` informa qual o atributo do produto que gerará o código de barras, o método `generate()` cria, e o `save()` salva as etiquetas em um arquivo. Posteriormente, abrimos uma janela de exibição.

```

public function onSend($param)
{
    try
    {
        // obtém os dados do formulário
        $data = $this->form->getData();
        $this->form->setData($data);

        // define as propriedades das etiquetas
        $properties['barcodeMethod'] = $data->method;
        $properties['leftMargin'] = 12;
        $properties['topMargin'] = 12;
        $properties['labelWidth'] = 64;
        $properties['labelHeight'] = 54;
        $properties['spaceBetween'] = 4;
        $properties['rowsPerPage'] = 5;
        $properties['colsPerPage'] = 3;
        $properties['fontSize'] = 12;
        $properties['barcodeHeight'] = 15;
        $properties['imageMargin'] = 0;

        // instancia o gerador de etiquetas
        $generator = new AdiantiBarcodeDocumentGenerator;
        $generator->setProperties($properties);
        $generator->setLabelTemplate($data->template);

        // lê os produtos da base de dados
        TTransaction::open('samples');
        $products = Product::all();

        // adiciona cada um dos produtos ao gerador
        foreach ($products as $product)
        {
            $product->barcode = str_pad($product->id, 10, '0', STR_PAD_LEFT);
            $product->description = substr($product->description, 0, 15);
            $generator->addObject($product);
        }

        // gera e salvar as etiquetas
        $generator->setBarcodeContent('barcode');
        $generator->generate();
        $generator->save('tmp/barcodes.pdf');

        // abre uma janela para apresentar o PDF final
        $window = TWindow::create(_t('Barcode'), 0.8, 0.8);
        $object = new TElement('object');
        $object->data = 'tmp/barcodes.pdf';
        $object->type = 'application/pdf';
        $object->style = "width: 100%; height:calc(100% - 10px)";

        // adiciona o objeto à janela
        $window->add($object);
        $window->show();

        // fecha transação
        TTransaction::close();
    }
    catch (Exception $e) {
        new TMessage('error', $e->getMessage());
    }
}
}

```

A figura a seguir demonstra o resultado da execução deste programa.

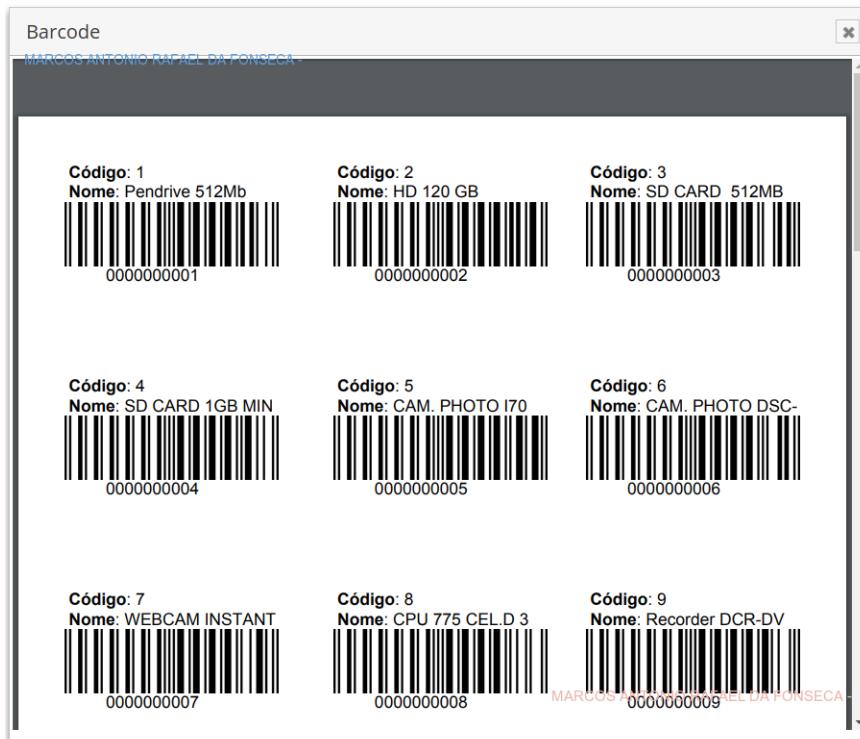


Figura 106 Etiquetas de Barcode

4.12.2 Etiquetas de QRCode

Para demonstrar a criação de etiquetas de QRCode, vamos utilizar o cadastro já existente de produtos, e apresentar um QRCode baseado em seu ID e descrição. Assim como já demonstrado nas etiquetas de códigos de barras, a classe `AdiantiBarcodeDocumentGenerator`, que é responsável pela geração das etiquetas de QRCode, permite que configuremos um modelo (template) para geração das etiquetas. Cada etiqueta pode conter outras informações além do QRCode, tais como códigos, descrições, etc.

Para criar as etiquetas de códigos de QRCode, vamos antes criar um formulário, onde o usuário poderá configurar o modelo (template), onde indicará quais informações deseja exibir nas etiquetas. A figura a seguir demonstra este formulário. No campo do template, o usuário poderá inserir campos como `{$id}`, mas também a posição onde entrará o código de QRCode com a marcação `#qrcode#`.

```

Template
<b>Código</b>: {$id}
<b>Nome</b>: {$description}
#qrkode#
{$id_pad}

Send
MARCOS ANTONIO RAFAEL DA FONSECA -

```

Figura 107 Formulário para geração de QRCode

O código fonte inicia com a montagem do formulário (`BootstrapFormBuilder`). Em seguida, é criado o campo para definição do Template de geração. A variável `$label` contém uma sugestão inicial de Template de geração, com alguns campos que temos a certeza que existem na tabela de produtos.

O formulário terá uma ação vinculada ao método `onSend()`, que gerará um PDF com os códigos de QRCode e apresentará este PDF em uma janela.

app/control/Presentation/Label/FormQrcodeView.class.php

```

<?php

class FormQrcodeView extends TPage
{
    private $form;

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

        // cria o formulário
        $this->form = new BootstrapFormBuilder;
        $this->form->setFormTitle(_t('QRCode'));

        // cria os campos do formulário
        $template = new TText('template');

        // adiciona os campos ao formulário
        $this->form->addFields( [new TLabel('Template')], [ $template ] );

        $template->setSize('80%', 100);

        $label = '' . "\n";
        $label .= '<b>Código</b>: {$id}' . "\n";
        $label .= '<b>Nome</b>: {$description}' . "\n";
        $label .= '#qrkode#' . "\n";
        $label .= '{$id_pad}';

        $template->setValue($label);

        // define a ação do formulário
        $this->form->addAction('Send', new TAction(array($this, 'onSend')), 'fa:check-circle-o green');

        parent::add($this->form);
    }
}

```

O método `onSend()` será executado a partir do botão de ação do formulário e gerará os códigos de QRCode. Este método inicia obtendo os dados do formulário, pelo método `getData()`, e em seguida define um vetor de propriedades da etiqueta. Dentre algumas propriedades estão: as margens da etiqueta, sua largura e altura, espaço entre etiquetas, e quantas linhas e colunas de etiquetas serão apresentadas por página.

A classe `AdiantiBarcodeDocumentGenerator` faz o processo de geração. Para tal, ela recebe as propriedades pelo método `setProperties()`, e o modelo de geração da etiqueta pelo método `setLabelTemplate()`. Por último, e mais importante, é necessário passar para a classe, cada um dos objetos que gerarão etiquetas, pois cada objeto gerará uma etiqueta correspondente. O método `addObject()` adiciona um objeto ao gerador de etiquetas. Os objetos neste caso são todos os produtos, que foram lidos pelo método `Product::all()`. Por fim, o método `setBarcodeContent()` informa qual o atributo do produto que gerará o QRCode, o método `generate()` cria, e o `save()` salva as etiquetas em um arquivo. Posteriormente, abrimos uma janela de exibição.

A figura a seguir demonstra o resultado deste programa.



Figura 108 Etiquetas de QRCode

4.12.3 Etiquetas em tela

Nos exemplos anteriores, vimos como gerar um documento PDF com etiquetas de códigos de barras e QRCodes. Entretanto, em alguns momentos é necessário apresentar as etiquetas em tela, não para impressão em PDF. Este exemplo demonstra como gerar dois tipos de códigos de barras e um QRCode utilizando as bibliotecas Picqer Barcode e Bacon QRCode, e apresentando seu resultado em tela.

Para gerar um código de barras em tela, utilizamos a classe `BarcodeGeneratorHTML` do pacote Picqer Barcode. O método `getBarcode()` retorna o código de barras no formato indicado (Ex: CODE_128, EAN_13).

O código de QRCode é criado pelas classes `Svg` e `Writer`, do pacote BaconQRCode. Após ajustarmos alguns tamanhos, utilizamos o método `writeString()` para gerar um código de QRCode com o conteúdo informado. Por fim, apresentamos os códigos gerados dentro de um `TPanelGroup`.

[app/control/Presentation/Label/FormScreenLabelsView.class.php](#)

```
<?php
class FormScreenLabelsView extends TPage
{
    public function __construct()
    {
        parent::__construct();

        // gera os barcodes
        $generator = new \Picqer\Barcode\BarcodeGeneratorHTML();

        // gera um código de barras CODE_128
        $bar1 = $generator->getBarcode('1231723897', $generator::TYPE_CODE_128, 5, 100);

        // gera um código de barras EAN_13
        $bar2 = $generator->getBarcode('1231723897', $generator::TYPE_EAN_13, 5, 100);

        // gera o qrcode
        $renderer = new \BaconQrCode\Renderer\Image\Svg();
        $renderer->setHeight(256);
        $renderer->setWidth(256);
        $renderer->setMargin(0);
        $writer = new \BaconQrCode\Writer($renderer);

        // gera o qrcode com essa string
        $qrCode = $writer->writeString('Hello World!');

        // empacota os códigos em um painel
        $panel = new TPanelGroup('Barcodes and QRCodes');
        $panel->add($bar1);
        $panel->add('<br>');
        $panel->add($bar2);
        $panel->add('<br>');
        $panel->add($qrCode);
        parent::add($panel);
    }
}
```

A figura a seguir demonstra o resultado deste programa.



Figura 109 Etiquetas em tela

CAPÍTULO 5

Organização e controle

Chegamos em um ponto do livro onde já aprendemos a manipular o banco de dados, e a criar interfaces com diversos componentes. Neste capítulo vamos criar interfaces mais completas que integram diversos componentes e que manipulam registros da base de dados. O objetivo deste capítulo é justamente abordar a criação de interfaces com formulários, listagens, buscas, e edições de registros na base de dados.

5.1 Cadastros padronizados

Vamos começar a abordar a criação de formulários e datagrids para manipulação de registros de forma ágil. Para tal, o Adianti Framework disponibiliza um conjunto de métodos padronizados para operações básicas. São métodos que podem ser utilizados como base para a criação de páginas como formulários e datagrids que já contém uma série de funcionalidades, facilitando muito a vida do desenvolvedor. Com isso, é possível criar um cadastro completo com poucas linhas de código.

5.1.1 Formulário padronizado

Para começar, vamos criar um formulário para cadastro e edição de registros de cidades (**City**). Este formulário terá campos para código, nome e estado. O formulário terá ainda um botão para “Save”, que obtém os dados do formulário e armazena-os na base de dados; um botão “Clear”, que limpa o formulário, preparando-o para um novo registro; e um botão “Listing”, que carregará adatagrid contendo os registros já cadastrados. Estadatagrid será construída no próximo exemplo. Na próxima figura, podemos ver o resultado da execução deste exemplo.

The screenshot shows a web-based form titled "Formulário manual". At the top left, it says "MARCOS ANTONIO RAFAEL DA FONSECA -". Below the title are three input fields: "ID" (text), "Name" (text), and "State" (dropdown). At the bottom of the form are three buttons: "Save" (green icon), "Clear" (red icon), and "Listing" (blue icon). To the right of the buttons, it says "MARCOS ANTONIO RAFAEL DA FONSECA -".

Figura 110 Formulário de cadastro padronizado

Para criar um formulário de maneira rápida, vamos nos basear em um conjunto de métodos preexistentes do Framework que fazem o trabalho de edição e gravação de registros de maneira automatizada. O único trabalho que teremos será o de construir o formulário no método construtor da classe e conectar os eventos do formulário a métodos pré-definidos pelo Framework.

Para “importar” métodos pré-definidos para uma classe, utilizamos o conceito de Trait. Um Trait contém um conjunto de métodos que pode ser incluído em uma ou mais classes. Alguns chamam Traits de “método inteligente para copy & paste”, uma vez que o Trait evita a necessidade de copiar e colar um determinado trecho de código entre diferentes classes, facilitando o reaproveitamento de uma funcionalidade.

Para importar um conjunto de métodos, utilizamos o operador “use” no início da declaração da classe, informando o caminho do Trait. O Trait `AdiantiStandardFormTrait` contém métodos como `onSave()`, `onClear()` e `onEdit()` pré-definidos.

A classe inicia pela chamada do método do Trait `setDatabase()`, que determina o nome da base de dados a ser utilizada pelo cadastro. Outro método importante do Trait é `setActiveRecord()`, que determinará qual o nome do Active Record que será manipulado pelo formulário. Com isso, basta criarmos um formulário.

Em seguida, criamos o formulário (`BootstrapFormBuilder`). Após a sua criação, são criados os campos (`$id`, `$name` e `$state_id`), e estes são adicionados ao formulário por meio do método `addField()`. Por fim, adicionamos as ações ao formulário: Salvar, que está vinculada ao método do Trait `onSave()`; Limpar, que está vinculada ao método do Trait `onClear()`; e Listar, que carrega a classe `StandardDataGridView`, que será construída no próximo exemplo e é responsável pela exibição dos registros.

Os métodos do Trait fazem somente operações básicas como obter os dados do formulário e gravar no banco de dados, e carregar os dados do banco de dados e apresentar no formulário. Sempre que precisarmos realizar transformações mais complexas nos registros durante os procedimentos de carregamento ou gravação, é recomendado optar por um cadastro manual, que será visto na próxima seção.

app/control/Organization/StandardControls/StandardFormView.class.php

```
<?php
class StandardFormView extends TPage
{
    protected $form; // form

    // importa trait com métodos onSave, onClear, onEdit
    use Adianti\Base\AdiantiStandardFormTrait;

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

        $this->setDatabase('samples');      // define o banco
        $this->setActiveRecord('City');     // define a classe

        $this->form = new BootstrapFormBuilder('form_City');
        $this->form->setFormTitle(_t('Standard Form'));

        // cria os campos
        $id      = new TEntry('id');
        $name    = new TEntry('name');
        $state_id = new TDBCombo('state_id', 'samples', 'State', 'id', 'name');
        $id->setEditable(FALSE);

        // adiciona os campos
        $this->form->addFields( [new TLabel('ID')], [$id] );
        $this->form->addFields( [new TLabel('Name'), 'red)], [$name] );
        $this->form->addFields( [new TLabel('State'), 'red)], [$state_id] );

        // adiciona validações
        $name->addValidation( 'Name', new TRequiredValidator);
        $state_id->addValidation( 'State', new TRequiredValidator);

        // define as ações do form
        $this->form->addAction('Save', new TAction(array($this, 'onSave')),
                               'fa:save green');
        $this->form->addActionLink('Clear', new TAction(array($this, 'onClear')),
                               'fa:eraser red');
        $this->form->addActionLink('Listing',
                               new TAction(array('StandardDataGridView', 'onReload')),
                               'fa:table blue');

        parent::add($this->form);
    }
}
```

5.1.2 Datagrid padronizada

Agora que já aprendemos a criar um formulário de maneira simples baseando-se em métodos padronizados fornecidos pelo Framework, vamos criar umadatagrid, que permita localização, exclusão e edição de registros, da mesma maneira.

Este exemplo será formado basicamente por um formulário e umadatagrid. A função do formulário, que ficará na parte superior, é buscar registros de cidade (**City**), por meio de seu nome. Este formulário também terá um botão “Novo”, cuja função é direcionar para a página de cadastro (construída no exemplo anterior). Já adatagrid, terá botões para edição e exclusão, bem como uma paginação de registros.

The screenshot shows a standard Adianti DataGrid interface. At the top, there is a header bar with the title "Datagrid padronizada" and a subtitle "MARCOS ANTONIO RAFAEL DA FONSECA -". Below the header is a search bar labeled "Name:" with a placeholder field. Underneath the search bar are two buttons: "Find" (with a magnifying glass icon) and "New" (with a plus sign icon). The main area contains a table with five rows of data. The columns are "Id", "Name", and "State". The data is as follows:

	Id	Name	State
	1	Lajeado	RS
	2	Porto Alegre	RS
	3	Caxias do Sul	RS
	4	São Paulo	SP
	5	Osasco	SP

At the bottom of the grid, there is a navigation bar with page numbers from 1 to 10, where page 1 is highlighted in blue. To the right of the navigation bar is the subtitle "MARCOS ANTONIO RAFAEL DA FONSECA -".

Figura 111 Datagrid de cadastro padronizada

Para automatizar adatagrid, vamos utilizar um Trait chamado `AdiantiStandardListTrait`. Este Trait contém, implementados de maneira genérica, os métodos mais utilizados na manipulação de datagrids, como métodos de busca e filtragem, carregamento de registros com paginação e exclusão.

Para “importar” métodos pré-definidos para uma classe, utilizamos o conceito de Trait. Um Trait contém um conjunto de métodos que pode ser incluído em uma ou mais classes. Alguns chamam Traits de “método inteligente para copy & paste”, uma vez que o Trait evita a necessidade de copiar e colar um determinado trecho de código entre diferentes classes, facilitando o reaproveitamento de uma funcionalidade.

Para importar um conjunto de métodos, utilizamos o operador “`use`” no início da declaração da classe, informando o caminho do Trait. O Trait `AdiantiStandardListTrait` contém métodos como `setDatabase()`, `setActiveRecord()`, `addFilterField()`, `onSearch()` e `onReload()`, `onDelete()`, `Delete()`, e outros pré-definidos.

A classe inicia pela chamada do método do Trait `setDatabase()`, que determina o nome da base de dados a ser utilizada pelo cadastro. Outro método importante do Trait é `setActiveRecord()`, que determinará qual o nome do Active Record que será manipulado pelo formulário. O método do Trait `addFilterField()` define como cada campo do formulário será utilizado para filtrar adatagrid. Já o método `setDefaultOrder()` define uma ordenação padrão para o carregamento inicial dadatagrid.

Em seguida, criamos o formulário de buscas(**BootstrapFormBuilder**). Após a sua criação, é criado o campo (`$name`), e este é adicionado ao formulário por meio do método `addField()`. O nome deste campo deve coincidir com o nome informado pelo método `addFilterField()`. Por fim, adicionamos as ações ao formulário: Buscar, que está vinculada ao método do Trait `onSearch()`; Novo, que está vinculada ao método `onClear()` da classe **StandardFormView**, que foi construída no exemplo anterior, e que abrirá um novo formulário para criação de registros. O método `setData()` é utilizado para manter o formulário preenchido com os dados da busca, que por sua vez são armazenados na variável de sessão `StandardDataGridView_filter_data`.

Outro método que pode ser utilizado neste caso é o `parent::setCriteria()` que definiria um filtro fixo para a listagem, ou seja, a datagrid seria carregada pré-filtrada.

Obs: Este exemplo usa sessões, que são manipuladas pela classe **TSession**. Sessões permitem reter valores de variáveis mesmo através de trocas de página. Para ler valores da sessão, basta utilizar o método `getValue('variavel')`, enquanto que para atribuir valores para a sessão, basta utilizar o método `setValue('variavel', 'valor')`.

Após construirmos o formulário de busca de registros, construímos uma datagrid (**TDataGrid**). Estadatagrid terá três colunas (código, nome e estado) adicionadas por meio do método `addColumn()`. Sobre as colunas de código e nome, aplicaremos uma ação (`setAction`) vinculada ao método `onReload()`, para provocar recarga com uma nova ordem. Assim, sempre que o usuário clicar sobre o título da coluna, este método é executado, identificando a coluna a ser utilizada para fins de ordenação.

Após adicionarmos as duas colunas dadatagrid, são criadas duas ações que podem ser executadas sobre as linhas dadatagrid. A ação de “Editar”, carregará o método `onEdit()` da classe **StandardFormView**, criada no exemplo anterior e passando o `id` como parâmetro. Desta forma, o formulário será carregado com base no atributo `id` do objeto presente nadatagrid. Já a ação “Deletar”, executará o método do Trait `onDelete()`, identificando o `id` do objeto. Este método, por sua vez, pergunta ao usuário se o mesmo deseja excluir o objeto, e em seguida excluirá o registro.

Logo em seguida, o modelo de dados dadatagrid é criado por meio do método `createModel()`. Este é o primeiro exemplo em que usamos o objeto **TPageNavigation**, responsável pela paginação dos dados. Este objeto deve ficar localizado abaixo dadatagrid e provê botões de navegação. O objeto **TPageNavigation** deve estar conectado também ao método de carga dadatagrid: `onReload()`. Por fim, é criado um container (**TVBox**) para conter os objetos verticalmente.

Obs: Os métodos padrão de datagrids pressupõem que o objeto que representa o formulário de busca seja `$this->form`, e adatagrid seja `$this->datagrid`. Então, ao utilizar um método padrão, você deve utilizar essa nomenclatura para esses objetos.

app/control/Organization/StandardControls/StandardDataGridView.class.php

```
<?php
class StandardDataGridView extends TPage
{
    protected $form;      // registration form
    protected $datagrid;  // listing
    protected $pageNavigation;

    // trait com onReload, onSearch, onDelete...
    use Adianti\Base\AdiantiStandardListTrait;

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

        $this->setDatabase('samples');           // define o banco
        $this->setActiveRecord('City');          // define a classe
        $this->addFilterField('name', 'like', 'name'); //campo, operador, campo do form
        $this->setDefaultOrder('id', 'asc');     // define a ordem padrão

        // cria o formulário
        $this->form = new BootstrapFormBuilder('form_search_City');
        $this->form->setFormTitle($t('Standard DataGridView'));

        $name = new TEntry('name');
        $this->form->addField($name);

        // define as ações
        $this->form->addAction('Find', new TAction([$this, 'onSearch']),
            'fa:search blue');
        $this->form->addActionLink('New', new TAction(['StandardFormView', 'onClear']),
            'fa:plus-circle green');

        // mantém o form preenchido com os dados da busca
        $this->form->setData(TSession::getValue('StandardDataGridView_filter_data'));

        // cria a datagrid
        $this->datagrid = new BootstrapDatagridWrapper(new TDataGrid);
        $this->datagrid->width = "100%";

        // cria as colunas dadatagrid
        $col_id    = new TDataGridColumn('id', 'Id', 'right', '10%');
        $col_name  = new TDataGridColumn('name', 'Name', 'left', '60%');
        $col_state = new TDataGridColumn('state->name', 'State', 'center', '30%');

        $this->datagrid->addColumn($col_id);
        $this->datagrid->addColumn($col_name);
        $this->datagrid->addColumn($col_state);

        // define as ações das colunas
        $col_id->setAction(new TAction([$this, 'onReload']), ['order' => 'id']);
        $col_name->setAction(new TAction([$this, 'onReload']), ['order' => 'name']);

        // cria ações de linha
        $action1 = new TDataGridAction(['StandardFormView', 'onEdit'], ['key' => '{id}']);
        $action2 = new TDataGridAction([$this, 'onDelete'], ['key' => '{id}']);

        $this->datagrid->addAction($action1, 'Edit', 'fa:edit blue');
        $this->datagrid->addAction($action2, 'Delete', 'fa:trash red');

        // cria o modelo (estrutura) dadatagrid em memória
        $this->datagrid->createModel();
    }
}
```

```

// cria o paginador de registros
$this->pageNavigation = new TPageNavigation;
$this->pageNavigation->setAction(new TAction(array($this, 'onReload')));
$this->pageNavigation->setWidth($this->datagrid->getWidth());

// empacota tudo usando um container
$ vbox = new TVBox;
$vbox->style = 'width: 100%';
$vbox->add($this->form);
$vbox->add(TPanelGroup::pack('', $this->datagrid, $this->pageNavigation));

// adiciona o container na página
parent::add($vbox);
}
}

```

5.1.3 Formulário com datagrid padronizada

Já vimos como construir formulários e datagrids individuais, com integração entre ambos, de maneira ágil. Estes recursos já nos abrem muitas possibilidades. Entretanto, em alguns casos, o cadastro que manipularemos é tão pequeno que não necessita de duas telas separadas (uma para formulário de cadastro e outra para listagem com busca) para realizar a manutenção dos registros. Quando sabemos que uma determinada tabela não passará de 20 ou 30 registros em toda sua existência, podemos ter toda a edição em uma mesma tela. A ideia básica de um controlador de formulário com datagrid é ter uma tela onde possamos ao mesmo tempo listar e editar os registros. Essa tela dispensa recursos como busca e paginação, uma vez que terá uma quantidade limitada de registros. Na próxima figura, podemos ver o resultado da execução deste exemplo.

The screenshot displays a web-based application interface. At the top, there is a header bar with the text "Form/datagrid padronizada" and "MARCOS ANTONIO RAFAEL DA FONSECA". Below the header, there is a form section containing two input fields: "ID" and "Name". Underneath the form are two buttons: "Salvar" (Save) and "Limpar" (Clear). In the bottom right corner of the form area, there is a small watermark-like text: "MARCOS ANTONIO RAFAEL DA FONSECA".

Below the form is a datagrid table with the following data:

	ID	Name
	1	Frequente
	2	Casual
	3	Varejista

At the bottom right of the datagrid, there is another watermark-like text: "MARCOS ANTONIO RAFAEL DA FONSECA".

Figura 112 Formulário com Datagrid padronizada

A ideia deste exemplo é termos uma página com um formulário e uma datagrid. O formulário será utilizado para cadastrar novos registros e adatagrid para exibir os registros já cadastrados. O formulário terá botões como “Salvar”, que obterá os dados digitados no formulário e armazenará na base de dados e “Limpar”, que limpará o formulário. Já adatagrid terá ações para editar e excluir.

Para criar este formulário comdatagrid de maneira rápida, vamos nos basear em um conjunto de métodos preexistentes do Framework que fazem o trabalho de edição e gravação de registros de maneira automatizada. O único trabalho que teremos será o de construir o formulário e adatagrid no método construtor da classe e conectar os eventos do formulário edatagrid a métodos pré-definidos pelo Framework.

Para “importar” métodos pré-definidos para uma classe, utilizamos o conceito de Trait. Um Trait contém um conjunto de métodos que pode ser incluído em uma ou mais classes. Alguns chamam Traits de “método inteligente para copy & paste”, uma vez que o Trait evita a necessidade de copiar e colar um determinado trecho de código entre diferentes classes, facilitando o reaproveitamento de uma funcionalidade.

Para importar um conjunto de métodos, utilizamos o operador “use” no início da declaração da classe, informando o caminho do Trait. O Trait `AdiantiStandardFormListTrait` contém métodos como `onSave()`, `onEdit()` e `onReload()` pré-definidos.

A classe inicia pela chamada do método do Trait `setDatabase()`, que determina o nome da base de dados a ser utilizada pelo cadastro. Outro método importante do Trait é `setActiveRecord()`, que determinará qual o nome do Active Record que será manipulado pelo formulário. O método `setDefaultOrder()` define a ordem padrão.

Em seguida, criamos o formulário (`BootstrapFormBuilder`). Após a sua criação, são criados os campos (`$id`, `$name`), e estes são adicionados ao formulário por meio do método `addField()`. Por fim, adicionamos as ações: Salvar, que está vinculada ao método `onSave()`; Limpar, que está vinculada ao método `onClear()`, ambos do Trait.

Após construirmos o formulário de cadastro, construímos umadatagrid (`TDataGrid`). Estadatagrid terá duas colunas (código, e nome) adicionadas por meio do método `addColumn()`. Sobre as colunas de código e nome, aplicaremos uma ação (`setAction`) vinculada ao método `onReload()`, para provocar recarga com uma nova ordem. Assim, sempre que o usuário clicar sobre o título da coluna, este método é executado, identificando a coluna a ser utilizada para fins de ordenação.

Após adicionarmos as duas colunas dadatagrid, são criadas duas ações que podem ser executadas sobre as linhas dadatagrid. A ação de “Editar”, carregará o método `onEdit()` passando o `id` como parâmetro, e “Deletar”, executará o método `onDelete()`, identificando o `id` do objeto. Este método, por sua vez, pergunta ao usuário se o mesmo deseja excluir o objeto, e em seguida excluirá o registro.

Obs: Os métodos padrão pressupõem que o objeto que representa o formulário de cadastro seja `$this->form`, e adatagrid seja `$this->datagrid`. Então, ao utilizar um método padrão, você deve utilizar essa nomenclatura para esses objetos na página.

app/control/Organization/StandardControls/StandardFormDataGridView.class.php

```
<?php
class StandardFormDataGridView extends TPage
{
    protected $form;           // formulário de cadastro
    protected $datagrid;        // listagem
    protected $loaded;          // carregado
    protected $pageNavigation;  // paginador

    // trait com onSave, onEdit, onDelete, onReload, onSearch...
    use Adianti\Base\AdiantiStandardFormListTrait;

    public function __construct()
    {
        parent::__construct();
        $this->setDatabase('samples'); // define o banco
        $this->setActiveRecord('Category'); // define a classe de modelo
        $this->setDefaultOrder('id', 'asc'); // define a ordem default
        $this->setLimit(-1); // desliga o limit

        // cria o formulário
        $this->form = new BootstrapFormBuilder('form_categories');
        $this->form->setFormTitle(_t('Standard Form/DataGrid'));

        // cria os campos
        $id      = new TEntry('id');
        $name   = new TEntry('name');

        // adiciona os campos ao formulário
        $this->form->addFields( [new TLabel('ID')], [ $id ] );
        $this->form->addFields( [new TLabel('Name', 'red')], [ $name ] );
        $name->addValidation('Name', new TRequiredValidator);

        // define as ações
        $this->form->addAction( 'Save', new TAction([$this, 'onSave']), 'fa:save green' );
        $this->form->addActionLink( 'Clear', new TAction([$this, 'onClear']), 'fa:eraser' );

        // desliga a edição
        $id->setEditable(FALSE);

        // cria adatagrid
        $this->datagrid = new BootstrapDatagridWrapper(new TDataGrid);
        $this->datagrid->width = '100%';

        // adiciona as colunas
        $col_id     = new TDataGridColumn('id', 'Id', 'right', '10%');
        $col_name   = new TDataGridColumn('name', 'Name', 'left', '90%' );
        $this->datagrid->addColumn($col_id);
        $this->datagrid->addColumn($col_name);

        // define ações de coluna
        $col_id->setAction( new TAction([$this, 'onReload']), ['order' => 'id']);
        $col_name->setAction( new TAction([$this, 'onReload']), ['order' => 'name']);

        // cria ações de linha
        $action1 = new TDataGridAction([$this, 'onEdit'], ['key' => '{id}']);
        $action2 = new TDataGridAction([$this, 'onDelete'], ['key' => '{id}']);
    }
}
```

```

// adiciona as ações de linha
$this->datagrid->addAction($action1, 'Edit', 'fa:edit blue');
$this->datagrid->addAction($action2, 'Delete', 'fa:trash red');

// cria o modelo dadatagrid em memória
$this->datagrid->createModel();

// empacota objetos usando uma caixa vertical
$ vbox = new TVBox;
$vbox->style = 'width: 100%';
$vbox->add($this->form);
$vbox->add(TPanelGroup::pack('', $this->datagrid));

// adiciona container à página
parent::add($vbox);
}

}

```

5.2 Cadastros manuais

Na seção anterior, aprendemos a criar formulários e datagrids integrados ao banco de dados de maneira ágil. A abordagem vista permite escrever programas que interagem com o banco de dados, com poucas linhas de código. Esta abordagem pode ser suficiente quando não é necessário realizar alguma transformação nos dados antes de armazená-los na base de dados, ou antes de apresentá-los na tela. Sempre que precisarmos realizar algum procedimento específico, é necessário criar os métodos para formulários e datagrids de forma manual. O objeto desta seção é demonstrar como criar esses cadastros de forma manual.

5.2.1 Formulário manual

Na seção anterior, criamos um formulário rápido, utilizando métodos do trait `AdiantiStandardFormTrait` para editar cidades. Esta abordagem nos permitiu criar um formulário com poucas linhas de código, porém sem poder algum de customização. Neste exemplo, recriaremos este controlador de formulários de forma manual, implementando os métodos que tinham sido generalizados como `onSave()` e `onEdit()`. Na próxima figura, podemos ver o resultado da execução deste exemplo.

The screenshot shows a web-based form titled "Formulário manual". At the top, it displays the user's name: "MARCOS ANTONIO RAFAEL DA FONSECA -". The form contains three input fields: "ID" (text), "Name" (text), and "State" (dropdown). Below the form are three buttons: "Save" (with a save icon), "Clear" (with a clear icon), and "Listing" (with a listing icon). The status bar at the bottom also shows the user's name: "MARCOS ANTONIO RAFAEL DA FONSECA -".

Figura 113 Formulário manual de cadastro

O formulário terá um botão “Save”, que obtém os dados do formulário e armazena-os na base de dados; um botão “Clear”, que limpa o formulário, preparando-o para um novo registro; e um botão “Listing”, que carregará adatagrid (construída no próximo exemplo), contendo os registros já cadastrados.

O exemplo, que pode ser acompanhado no código-fonte a seguir, inicia com a criação do formulário (`BootstrapFormBuilder`). Em seguida, são criados os campos do formulário: `id`, `name`, e `state_id`. Após, acrescentamos os campos ao formulário.

Após a adição dos campos na tabela, é criado um botão de ação “Save”, que está conectado ao método `onSave()`. Este método será responsável por obter os dados do formulário e salvá-los na base de dados. Em seguida, é criado um botão de ação “Clear”, que está conectado ao método `onClear()`. Após, é criado um botão para exibir a listagem de registros, que carregará a classe `CompleteDataGridView`, a qual será criada no próximo exemplo. Por fim, o formulário é adicionado à página.

`app/control/Organization/ManualControls/CompleteFormView.class.php`

```
<?php
class CompleteFormView extends TPage
{
    private $form;
    function __construct()
    {
        parent::__construct();

        // cria o formulário
        $this->form = new BootstrapFormBuilder('form_City');
        $this->form->setFormTitle(_t('Manual form'));

        // cria os campos do formulário
        $id      = new TEntry('id');
        $name    = new TEntry('name');
        $state_id = new TDBCombo('state_id', 'samples', 'State', 'id', 'name');
        $id->setEditable(FALSE);

        // adiciona os campos ao formulário
        $this->form->addFields( [new TLabel('ID')], [ $id ] );
        $this->form->addFields( [new TLabel('Name', 'red')], [ $name ] );
        $this->form->addFields( [new TLabel('State', 'red')], [ $state_id ] );

        $name->addValidation('Name', new TRequiredValidator);
        $state_id->addValidation('State', new TRequiredValidator);

        // define a ação do formulário
        $this->form->addAction('Save', new TAction([$this, 'onSave']), 'fa:save green');
        $this->form->addActionLink('Clear',
            new TAction([$this, 'onClear']), 'fa:eraser');
        $this->form->addActionLink('Listing',
            new TAction(['CompleteDataGridView', 'onReload']), 'fa:table blue');

        parent::add($this->form);
    }
}
```

Agora que já criamos o layout do formulário no método construtor, procedemos com a declaração dos métodos que responderão aos botões de ação. Em primeiro lugar, o botão salvar está conectado ao método `onSave()`. Este método deve coletar os dados do formulário e armazenar o registro na base de dados.

O método `onSave()` abre uma transação com a base de dados, obtém os dados do formulário por meio do método `getData()`, que retorna um objeto (`$data`). Em seguida instanciamos um objeto `City` vazio, e preenchemos ele com o método `fromArray()`, a partir dos dados enviados pelo formulário. Após, o método `store()` é executado para gravar os dados na base de dados. Caso fosse necessário realizar alguma transformação no objeto antes de salvá-lo, esta transformação deveria ser realizada antes do método `store()`. Em seguida, os dados são devolvidos ao formulário pelo método `setData()`. A transação é fechada e uma mensagem é exibida ao usuário.

```
function onSave()
{
    try {
        TTransaction::open('samples'); // abre transação
        $this->form->validate(); // roda as validações de formulário
        $data = $this->form->getData(); // obtém os dados digitados

        $object = new City; // cria um objeto vazio
        $object->fromArray( (array) $data); // preenche os atributos
        $object->store(); // armazena o objeto

        // preenche o formulário com os dados do objeto
        $this->form->setData($object);

        TTransaction::close(); // fecha a transação

        new TMessage('info', 'Record saved');
    }
    catch (Exception $e) {
        new TMessage('error', $e->getMessage()); // exibe mensagem de exceção
        $this->form->setData( $this->form->getData() ); // preenche form
        TTransaction::rollback(); // desfaz operações da transação
    }
}
```

Obs: Após a chamada do método `store()`, o objeto Active Record recebe um `ID`. Desta forma, quando o método `setData()` é executado, o objeto já possui um `ID`, que é exibido no formulário. A partir desse momento, caso o usuário clique novamente em “Salvar”, o registro será alterado e não inserido como novo.

O método `onClear()` será executado com o objetivo de limpar o formulário.

```
public function onClear()
{
    $this->form->clear( TRUE );
}
```

Este formulário precisa ter um método de edição. A principal função do método de edição é receber um `ID` e preencher o formulário com os dados do respectivo objeto. Geralmente, o método de edição é executado a partir dadatagrid, no momento em que o usuário clica sobre um registro para editá-lo. Entretanto, o método de edição também pode ser utilizado para limpar o formulário (botão “Novo”), caso não seja identificado o registro a ser editado.

No Framework, os métodos de edição normalmente são chamados `onEdit()`, assim como os de salvamento `onSave()`. O método `onEdit()` verifica se é informado um parâmetro '`id`', que identifica o registro a ser editado. Esta informação geralmente vem de umadatagrid (criada no próximo exemplo).

Caso o método `onEdit()` seja executado puramente, sem identificar registro algum, o formulário é limpo por meio do método `clear()`. Caso a chave do registro seja informada, é aberta uma transação com a base de dados, e é instanciado o objeto Active Record (`City`), com base na chave informada (`$key`). Em seguida, os dados do Active Record são usados para preencher o formulário, e a transação é finalizada. Assim, o formulário é preenchido com os dados do objeto.

```
function onEdit($param)
{
    try {
        if (isset($param['id'])) // verifica se o parâmetro foi informado
        {
            $key = $param['id']; // obtém a chave do registro
            TTransaction::open('samples'); // abre a transação
            $object = new City($key); // instancia o Active Record
            $this->form->setData($object); // preenche o formulário
            TTransaction::close(); // fecha a transação
        }
        else {
            $this->form->clear( true ); // limpa o formulário
        }
    }
    catch (Exception $e) {
        new TMessage('error', $e->getMessage());
        TTransaction::rollback(); // desfaz operações
    }
}
```

Obs: Todos os métodos de manipulação de base de dados, como `onSave()` e `onEdit()`, devem realizar as operações dentro de um bloco de controle de exceções `try/catch`, pois as classes de manipulação da base de dados lançam exceções em caso de erros.

5.2.2 Datagrid manual

Agora que entendemos como construir um formulário completo, sem métodos padrão do Framework, faremos o mesmo com umadatagrid. O objetivo deste exemplo é construir umadatagrid completa, com busca, edição e exclusão de registros de cidades (`City`), com os métodos totalmente programados manualmente. Na próxima figura, veremos o resultado da execução deste exemplo.

The screenshot shows a web application interface for managing cities. At the top, there is a header with the title "Datagrid manual" and a subtitle "MARCOS ANTONIO RAFAEL DA FONSECA -". Below the header is a search bar with the placeholder "Name:" and two buttons: "Find" (with a magnifying glass icon) and "New" (with a plus sign icon). The main area contains a table with columns "Id", "Name", and "State". The data in the table is:

	Id	Name	State
	1	Lajeado	RS
	2	Porto Alegre	RS
	3	Caxias do Sul	RS
	4	São Paulo	SP
	5	Osasco	SP

At the bottom of the table is a navigation bar with numbered buttons from 1 to 10, where the first button is highlighted in blue. To the right of the navigation bar is the text "MARCOS ANTONIO RAFAEL DA FONSECA -".

Figura 114 Datagrid de cadastro manual

Este exemplo será formado basicamente por um formulário e uma datagrid. A função do formulário é buscar registros de cidade (`City`), por meio de seu nome. Este formulário também terá um botão “Novo”, cuja função é direcionar para a página de cadastro (construída no último exemplo). Já a datagrid terá botões para edição e exclusão, bem como uma paginação de registros.

Como pode ser visto no próximo código-fonte, o formulário de buscas, bem como a datagrid, são criados no método construtor. Neste método, criamos o formulário (`BootstrapFormBuilder`), e em seguida, é criado o campo utilizado para buscas (`TEntry`). Após, este campo é acrescentado ao formulário e é criada uma ação chamada “Find”, vinculada ao método `onSearch()`, que será responsável por filtrar a datagrid. Também é criado o botão “New”, que é vinculado ao método `onClear()` da classe `CompleteFormView`, fazendo com que o formulário de edição de registros seja carregado quando o usuário clicar nesse botão.

Após a criação do formulário de buscas, é criada a datagrid (`TDataGrid`). Inicialmente são criadas as colunas da datagrid (`TGridColumn`) e duas ações vinculadas às colunas da datagrid por meio do método `setAction()`. O objetivo destas ações é permitir o reordenamento da datagrid. Para isso, essas ações são vinculadas ao método `onReload()`, passando o parâmetro `order`, representando o nome da coluna.

A datagrid terá duas ações (`TDataGridAction`): `$action1` e `$action2`. A primeira ação tem como objetivo editar o registro da datagrid. Para isso, ela está vinculada ao método `onEdit()`, da classe `CompleteFormView`, construída no exemplo anterior. A segunda ação tem como objetivo excluir o registro, e está vinculada ao método `onDelete()` da mesma classe. Ambas ações recebem o `ID` do objeto como parâmetro.

Após criar adatagrid, é criado o navegador de páginas (`TPageNavigation`). Este objeto cria uma barra de paginação. Para tal, ele deve estar vinculado ao método de carregamento de registros, que no caso deste exemplo, é o método `onReload()`. Após criar todos os objetos, os mesmos são “empacotados” em uma caixa vertical (`$vbox`), para então serem adicionados à página.

app/control/Organization/ManualControls/CompleteDataGridView.class.php

```
<?php
class CompleteDataGridView extends TPage
{
    private $form, $datagrid, $pageNavigation, $loaded;
    public function __construct()
    {
        parent::__construct();

        // cria o formulário de buscas
        $this->form = new BootstrapFormBuilder('form_search_City');
        $this->form->setFormTitle(_t('Manual DataGrid'));

        $name = new TEntry('name');
        $this->form->addFields( [new TLabel('Name:'), [$name] ] );

        // cria as ações do formulário
        $this->form->addAction('Find', new TAction([$this, 'onSearch']), 'fa:search blue');
        $this->form->addActionLink('New', new TAction(['CompleteFormView',
            'onClear']), 'fa:plus-circle green');

        // mantém o formulário preenchido com os dados de busca
        $name->setValue( TSession::getValue( 'City_name' ) );

        // cria adatagrid
        $this->datagrid = new BootstrapDatagridWrapper(new TDataGrid);
        $this->datagrid->width = '100%';

        // cria as colunas dadatagrid
        $col_id      = new TDataGridColumn('id', 'Id', 'right', '10%');
        $col_name    = new TDataGridColumn('name', 'Name', 'left', '60%');
        $col_state   = new TDataGridColumn('state->name', 'State', 'center', '30%');

        // define as ações do clique na coluna
        $col_id->setAction(new TAction([$this, 'onReload']), ['order' => 'id']);
        $col_name->setAction(new TAction([$this, 'onReload']), ['order' => 'name']);

        // adiciona as colunas nadatagrid
        $this->datagrid->addColumn($col_id);
        $this->datagrid->addColumn($col_name);
        $this->datagrid->addColumn($col_state);

        // cria as ações dadatagrid
        $action1 = new TDataGridAction(['CompleteFormView', 'onEdit'], ['key' => '{id}']);
        $action2 = new TDataGridAction([$this, 'onDelete'], ['key' => '{id}']);

        $this->datagrid->addAction($action1, 'Edit', 'fa:edit blue');
        $this->datagrid->addAction($action2, 'Delete', 'fa:trash red');

        // cria a estrutura dadatagrid em memória
        $this->datagrid->createModel();
    }
}
```

```

// cria a paginação dadatagrid
$this->pageNavigation = new TPageNavigation;
$this->pageNavigation->setAction(new TAction([$this, 'onReload']));
$this->pageNavigation->setWidth($this->datagrid->getWidth());

// empacota os objetos em uma caixa vertical
$ vbox = new TVBox;
$vbox->style = 'width: 100%';
$vbox->add($this->form); // add a row to the form
$vbox->add(TPanelGroup::pack('', $this->datagrid, $this->pageNavigation));

// adiciona o container à página
parent::add($vbox);
}

```

Acima dadatagrid temos um formulário de buscas com um botão de ação vinculado ao método `onSearch()`. Este método obtém os dados do formulário de buscas, por meio do método `getData()`, e verifica se o usuário preencheu o campo de buscas (`name`). Caso preenchido, é criado um filtro (`TFilter`) e este é armazenado na sessão para posteriormente ser utilizado no método `onReload()`, responsável pelo carregamento dos objetos. Mas qual a razão de manter o filtro na sessão? O filtro é mantido na sessão pois adatagrid se mantém filtrada mesmo após sucessivas recargas de página, como na paginação dadatagrid. O método `onReload()` é executado ao final.

```

function onSearch()
{
    // obtém os dados preenchidos
    $data = $this->form->getData();

    // verifica se o usuário preencheu o campo de busca
    if (isset($data->name))
    {
        // cria um filtro com mo que foi digitado
        $filter = new TFilter('name', 'like', "%{$data->name}%");

        // grava o filtro e o conteúdo digitado na sessão
        TSession::setValue('City_filter', $filter);
        TSession::setValue('City_name', $data->name);

        // mantém o formulário preenchido
        $this->form->setData($data);
    }

    $param = array();
    $param['offset'] = 0;
    $param['first_page'] = 1;
    $this->onReload($param);
}

```

Obs: Ao executar o método `onSearch()`, para buscar registros, adatagrid sempre volta para a primeira página, o que pode ser visto no método `onSearch()`, ao tornar o parâmetro '`offset`' igual a 0, e o parâmetro '`first_page`', igual a 1.

O método `onReload()` é programado para ser executado sempre que a página for exibida, bem como quando o usuário realiza buscas. Este método deve carregar na datagrid os registros de cidade (`City`). Estadatagrid deverá ser paginada e carregar 10 registros de cada vez. O método inicia com a abertura da transação com a base de dados. Posteriormente, é criado um critério de seleção de registros. O método `setProperties()` da classe `TCriteria` define algumas variáveis dinâmicas como ordenamento (`order`) e `offset` da seleção, que são parâmetros que vêm da paginação.

Caso exista algum filtro armazenado na sessão (`City_filter`), este é adicionado ao critério de seleção. Assim, os filtros realizados pelo botão de busca, sempre são levados em consideração no momento de carregar adatagrid.

Os objetos são carregados pelo método `load()`, e depois, inseridos nadatagrid por meio do método `addItem()`. Em seguida, contamos quantos objetos seriam carregados nadatagrid ao total se não tivéssemos paginação. Isto é importante para sabermos quantas “páginas” o paginador terá. Para contarmos o total de registros, usamos o método `count()`, mas antes precisamos executar o método `resetProperties()` para que o critério de seleção de registros não use `limit`, `order` e `offset`, o que influenciaria na contagem. Após saber o total de registros, configuramos o paginador com atributos como a quantidade de registros total, a ordenação e o limite de registros por página. Assim, o paginador terá condições de ser renderizado corretamente.

```
function onReload($param = NULL)
{
    try
    {
        TTransaction::open('samples'); // abre transação

        // instancia um repositório
        $repository = new TRepository('City');
        $limit = 10;

        // instancia um critério
        $criteria = new TCriteria;

        // define uma ordem default
        if (empty($param['order']))
        {
            $param['order'] = 'id';
            $param['direction'] = 'asc';
        }

        // configura o critério com base nos parâmetros da URL
        $criteria->setProperties($param); // order, offset
        $criteria->setProperty('limit', $limit);

        // verifica se usuário preencheu um filtro
        if (TSession::getValue('City_filter'))
        {
            // adiciona o filtro ao critério de seleção
            $criteria->add(TSession::getValue('City_filter'));
        }

        // carrega os objetos conforme o critério de seleção
        $objects = $repository->load($criteria);
    }
}
```

```

$this->datagrid->clear();

if ($objects)
{
    // percorre os objetos
    foreach ($objects as $object)
    {
        // adiciona o objeto nadatagrid
        $this->datagrid->addItem($object);
    }
}

// reset nos critérios (limit, offset)
$criteria->resetProperties();
$count = $repository->count($criteria);

$this->pageNavigation->setCount($count); // qtde registros
$this->pageNavigation->setProperties($param); // order, page
$this->pageNavigation->setLimit($limit); // limit

// fecha transação
TTransaction::close();
$this->loaded = true;
}
catch (Exception $e) {
    new TMessage('error', $e->getMessage()); // exibe exceção
    TTransaction::rollback(); // desfaz operações da transação
}
}

```

Adatagrid possui basicamente dois botões de ação: “Editar”, que está vinculado diretamente ao método `onEdit()` da classe `CompleteFormView`, e “Excluir”, que está vinculado ao método `onDelete()`.

A ação de excluir registros foi dividida em dois métodos: `onDelete()`, que questiona o usuário se ele deseja realmente excluir o registro; e `Delete()`, que efetua a exclusão. O método `onDelete()` basicamente monta um diálogo de questionamento (`TQuestion`). Este diálogo de questionamento está vinculado à ação `$action`, que simplesmente repassará os parâmetros recebidos (`$param`) para a próxima ação (`Delete`), visto que dentre os parâmetros está o código a ser excluído.

Caso o usuário responda afirmativamente, o método `Delete()` é executado, recebendo a chave do registro a ser excluído. O método basicamente carrega o objeto cidade (`City`) dentro de uma transação com base na chave do registro (`$key`). Em seguida executa o seu método `delete()`.

```

public static function onDelete($param)
{
    // cria a ação para resposta positiva
    $action = new TAction(array(__CLASS__, 'Delete'));
    $action->setParameters($param); // repassa parâmetros recebidos para a ação

    // exibe o diálogo para o usuário
    new TQuestion('Deseja excluir o registro?', $action);
}

```

```

public static function Delete($param)
{
    try
    {
        $key=$param['key']; // obtém a chave do registro
        TTransaction::open('samples'); // abre transação
        $object = new City($key, FALSE); // carrega objeto
        $object->delete(); // exclui objeto
        TTransaction::close(); // fecha transação

        // define ação posterior à confirmação
        $pos_action = new TAction([__CLASS__, 'onReload']);
        new TMessage('info', 'Registro excluído', $pos_action);
    }
    catch (Exception $e)
    {
        new TMessage('error', $e->getMessage()); // mensagem de exceção
        TTransaction::rollback(); // desfaz operações da transação
    }
}

```

Obs: Após o registro ser excluído, é exibida uma mensagem de confirmação (**TMessage**). Esta mensagem possui uma ação posterior (**\$pos_action**). Assim, quando o usuário confirmar a mensagem, será redirecionado novamente ao método **onReload()**, provocando a recarga da datagrid. Assim, o usuário visualizará adatagrid atualizada após a exclusão.

Por fim, temos o método **show()**, que é sobreescrito para executar o método **onReload()**, antes de exibir a página.

```

function show()
{
    if (!$this->loaded) {
        $this->onReload('func_get_arg(0)');
    }
    parent::show();
}

```

Obs: Sempre que sobreescrivemos um método, tal como o método **show()** nesse exemplo, é importante executar o método da classe pai. Caso contrário, a funcionalidade padrão não será executada. Neste caso, a página não seria exibida.

5.2.3 Formulário com datagrid manual

Já vimos como construir formulários e datagrids com controladores manuais. Como vimos na seção anterior, em alguns casos o cadastro que manipularemos é tão pequeno que não necessita de duas telas separadas (uma para formulário de cadastro e outra para listagem com busca) para realizar a manutenção dos registros, podemos ter toda a edição em uma mesma tela. O objetivo desse exemplo é criar um controlador manual de formulário com datagrid, onde teremos uma tela, onde possamos ao mesmo tempo, listar e editar os registros. Na próxima figura, podemos ver o resultado da execução deste exemplo.

Form/datagrid manual
MARCOS ANTONIO RAFAEL DA FONSECA -

	ID	Name
<input checked="" type="checkbox"/>	1	Frequente
<input checked="" type="checkbox"/>	2	Casual
<input checked="" type="checkbox"/>	3	Varejista

MARCOS ANTONIO RAFAEL DA FONSECA -

Figura 115 Formulário de cadastro comdatagrid manual

A ideia deste exemplo é termos uma página com um formulário e umadatagrid. O formulário será utilizado para cadastrar novos registros e adatagrid para exibir os registros cadastrados. O formulário terá botões como “Save”, que obterá os dados digitados no formulário e armazenará na base de dados e “Clear”, que limpará o formulário. Já adatagrid terá ações para editar e excluir os registros.

O exemplo inicia com a criação do formulário que será utilizado para a edição dos registros (`BootstrapFormBuilder`). Logo em seguida são criados os campos do formulário e estes são adicionados ao formulário pelo método `addField()`. O formulário terá duas ações: “Save”, que está conectada ao método `onSave()`, responsável por salvar o registro; e “Clear”, que está conectado ao método `onClear()`, responsável por limpar o formulário.

Logo apóis criarmos o formulário, é criada adatagrid (`TDataGrid`). Em seguida, são adicionadas duas colunas nadatagrid por meio do método `addColumn()`. As colunas estarão vinculadas ao método `onReload()`, que é executado quando o usuário clicar sobre o título da coluna, passando o parâmetro ‘`order`’. Também são adicionadas duas ações por meio do método `addAction()`, que são: “Edit”, que está conectada ao método `onEdit()`, e passa como parâmetro o id do objeto a fim de editá-lo; e “Delete”, que está conectada ao método `onDelete()`, e também passa o id do objeto como parâmetro a fim de excluí-lo. Ao fim do método construtor, os objetos são empacotados em uma caixa vertical (`TVBox`) e esta, é adicionada à página.

app/control/Organization/ManualControls/CompleteFormDataGridView.class.php

```
<?php
class CompleteFormDataGridView extends TPage
{
    private $form;
    private $datagrid;
    private $loaded;
```

```

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

    $this->form = new BootstrapFormBuilder('form_categories');
    $this->form->setFormTitle(_t('Manual Form/DataGrid'));

    // cria os campos do formulário
    $id      = new TEntry('id');
    $name   = new TEntry('name');

    // adiciona os campos ao formulário
    $this->form->addFields( [new TLabel('ID')], [ $id ] );
    $this->form->addFields( [new TLabel('Name'), 'red'], [ $name ] );

    $name->addValidation('Name', new TRequiredValidator);

    // define as ações do formulário
    $this->form->addAction( 'Save', new TAction([$this, 'onSave']), 'fa:save' );
    $this->form->addActionLink( 'Clear', new TAction([$this, 'onClear']),
        'fa:eraser' );

    // define ID como não-editável
    $id->setEditable(FALSE);

    $this->datagrid = new BootstrapDatagridWrapper(new TDataGrid);
    $this->datagrid->width = '100%';

    // cria as colunas da datagrid
    $col_id   = new TDataGridColumn('id', 'Id', 'right', '10%' );
    $col_name = new TDataGridColumn('name', 'Name', 'left', '90%' );

    $this->datagrid->addColumn($col_id);
    $this->datagrid->addColumn($col_name);

    // define as ações das colunas
    $col_id->setAction( new TAction([$this, 'onReload']), ['order' => 'id']);
    $col_name->setAction( new TAction([$this, 'onReload']), ['order' => 'name']);

    // cria as ações da datagrid
    $action1 = new TDataGridAction([$this, 'onEdit'], ['key' => '{id}']);
    $action2 = new TDataGridAction([$this, 'onDelete'], ['key' => '{id}']);

    $this->datagrid->addAction($action1, 'Edit', 'fa:edit blue');
    $this->datagrid->addAction($action2, 'Delete', 'fa:trash red');

    // cria estrutura dadatagrid em memória
    $this->datagrid->createModel();

    // empacota objetos usando uma caixa vertical
    $vbox = new TVBox;
    $vbox->style = 'width: 100%';
    $vbox->add($this->form);
    $vbox->add(TPanelGroup::pack('', $this->datagrid));

    // adiciona o container na página
    parent::add($vbox);
}

```

O método `show()` da página é redefinido para executar o método `onReload()`. Assim, sempre que a página for exibida, antes o método `onReload()`, será executado a fim de carregar os registros na datagrid.

```

function show()
{
    if (!$this->loaded)
    {
        // recarrega adatagrid, preservando possíveis parâmetros (limit, offset)
        $this->onReload( func_get_arg(0) );
    }
    parent::show();
}

```

O método `onReload()` abre uma transação com a base de dados e carrega todos os objetos da classe `Category`, adicionando-os à datagrid por meio do método `addItem()`. Neste exemplo, não teremos paginação (`TPageNavigation`), já que estamos construindo um formulário com datagrid a fim de editar tabelas com poucos registros. Desta maneira, carregaremos sempre toda a coleção de objetos a fim de exibir na datagrid. Como não temos paginação, também não precisamos nos preocupar com questões relativas a `limit` e `offset`.

```

function onReload($param = NULL)
{
    try {
        // abre a transação
        TTransaction::open('samples');

        $order = isset($param['order']) ? $param['order'] : 'id';

        // carrega todos objetos conforme a ordem
        $categories = Category::orderBy($order)->load();

        $this->datagrid->clear();
        if ($categories)
        {
            // percorre os objetos
            foreach ($categories as $category)
            {
                // adiciona o objeto nadatagrid
                $this->datagrid->addItem($category);
            }
        }
        // fecha transação
        TTransaction::close();
        $this->loaded = true;
    }
    catch (Exception $e) {
        new TMessage('error', $e->getMessage());
        TTransaction::rollback();
    }
}

```

O formulário de edição presente nesta página possui um botão “Salvar”, que está vinculado ao método `onSave()`. Na medida em que o usuário submete o formulário, este método deve obter os dados do formulário e armazená-los. Para tal, o método `onSave()` abre uma transação com a base de dados na forma de um Active Record `Category`, armazena este objeto, e então, fecha a transação com a base de dados.

Obs: É importante lembrar que no momento em que estivermos editando um registro, o campo `id` estará preenchido no formulário. Desta forma, o método `store()` procederá com um `update` (atualização de registro), caso contrário, com um `insert` (inserção de registro).

```

function onSave()
{
    try
    {
        TTransaction::open('samples'); // abre transação
        $this->form->validate(); // roda validações

        // obtém os dados do formulário como um Active Record Category
        $category = $this->form->getData('Category');
        $category->store(); // armazena o objeto
        TTransaction::close(); // fecha transação

        new TMessage('info', 'Registro salvo'); // mensagem de sucesso
        $this->onReload(); // recarrega listagem
    }
    catch (Exception $e)
    {
        new TMessage('error', $e->getMessage());
        TTransaction::rollback(); // desfaz operações
    }
}

```

Todo formulário tem no mínimo duas ações: salvar e editar. Assim como temos `onSave()` para salvar, devemos ter um método `onEdit()` para editar o formulário. A ação de edição é conectada geralmente a partir da datagrid, a fim de carregar o formulário com os dados do objeto a ser editado. O método `onEdit()` recebe um vetor de parâmetros e coleta a posição '`id`', que contém a chave do registro, passada a partir dadatagrid. Com base nesta chave, carrega o objeto (`Category`) e preenche o formulário de edição com os dados deste objeto, por meio do método `setData()`.

```

function onEdit($param)
{
    try
    {
        if (isset($param['id']))
        {
            $key = $param['id']; // obtém a chave do registro
            TTransaction::open('samples'); // abre transação
            $category = new Category($key); // instancia o Active Record Category
            $this->form->setData($category); // lança os dados do objeto no form
            TTransaction::close(); // fecha transação
            $this->onReload(); // recarrega a listagem
        }
        else
        {
            $this->form->clear( true );
        }
    }
    catch (Exception $e) // em caso de exceção
    [
        new TMessage('error', $e->getMessage());
        TTransaction::rollback(); // desfaz as operações
    ]
}

```

O método `onClear()` será executado com o objetivo de limpar o formulário.

```
public function onClear()
{
    $this->form->clear( true );
}
```

Para finalizar este programa, temos o método `onDelete()`, conectado a partir dadatagrid. Este método é executado sempre que o usuário solicita a exclusão de um registro, clicando sobre o ícone correspondente. Neste caso, ele solicita ao usuário se o mesmo deseja realmente excluir o registro selecionado. Aqui, é importante notar que é criada uma ação (`TAction`), que simplesmente repassa adiante os parâmetros recebidos por ela por meio do método `setParameters()` para a próxima ação (`Delete`). Dentre os parâmetros, está um chamado '`id`', que contém a chave do registro, que será utilizado para encontrá-lo.

```
public static function onDelete($param)
{
    // cria uma ação
    $action = new TAction(__CLASS__, 'Delete');

    // passa os parâmetros adiante
    $action->setParameters($param);

    // diálogo de questionamento
    new TQuestion('Deseja realmente excluir o registro?', $action);
}
```

O método `Delete()` é executado a partir do questionamento em `onDelete()`. O seu objetivo é receber a chave do registro '`id`', carregar o Active Record correspondente (`Category`) com base na chave, e excluir o objeto da base de dados, o que é realizado pelo método `delete()`. Após exclusão, a transação é fechada e uma mensagem é exibida ao usuário com a classe `TMessage`. Após o mesmo confirmar a mensagem, uma ação é executada (`$pos_action`). Esta ação recarrega a página atual para atualizar adatagrid.

```
public static function Delete($param)
{
    try {
        $key = $param['id']; // obtém a chave do registro

        TTransaction::open('samples'); // abre transação
        $category = new Category($key); // carrega o objeto
        $category->delete(); // deleta o objeto
        TTransaction::close(); // fecha transação

        $pos_action = new TAction(__CLASS__, 'onReload');
        new TMessage('info', 'Registro excluído', $pos_action);
    }
    catch (Exception $e) {
        new TMessage('error', $e->getMessage());
        TTransaction::rollback(); // desfaz operações
    }
}
```

5.3 Cadastros completos

O objetivo dessa seção, é apresentar exemplos de cadastros mais elaborados em relação àqueles já apresentadas até então. Para tal, elaboraremos telas para cadastros de produtos e clientes.

5.3.1 Formulário de clientes

Nesta seção, criaremos um formulário para manipular clientes, bem como dados relacionados a ele, como contatos e habilidades. Até o momento vimos diferentes abordagens para criar formulários e listagens. Entretanto, até o momento criamos formulários para manipular tabelas simples, sem muitos relacionamentos. Esta abordagem foi proposital, uma vez que precisávamos ser didáticos em alguns pontos.

Mas o que faltava era criar uma tela na qual pudéssemos reunir uma série de conceitos vistos até o momento. O formulário de manipulação de clientes é um ótimo exemplo para isso, pois é um cadastro que integra diferentes tipos de componentes (**TEntry**, **TDate**, **TDBUniqueSearch**, **TRadioGroup**, **TCombo**, **TDBCombo**, etc), o que por si só já torna este exemplo bastante atrativo.

Não bastasse isso, a classe de clientes (**Customer**) é a classe do nosso modelo que mais possui relacionamentos. Veja na lista a seguir os relacionamentos que ela possui, o que pode ser conferido na figura logo após a lista.

- Um objeto **Customer** (cliente) está associado a um objeto do tipo **Category** (categoria). Categoria pode ser: frequente, casual, varejista;
- Um objeto **Customer** (cliente) está associado a um objeto **City** (cidade);
- Um objeto **Customer** (cliente) possui uma composição com nenhum ou vários objetos **Contact** (contato). Isto significa que um objeto contato (objeto parte) é parte de somente um objeto cliente (objeto todo);
- Um objeto **Customer** (cliente) possui uma agregação com nenhum ou vários objetos **Skill** (habilidade). Assim, um objeto habilidade (objeto parte) pode ser parte de diferentes objetos cliente (objeto todo).

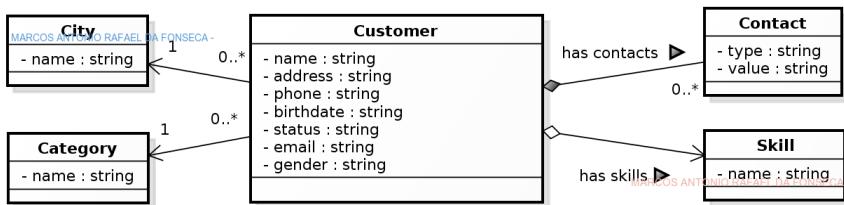


Figura 116 – Modelo de classes ao redor de Customer

Um formulário para manipulação de clientes deve reunir todos esses conceitos (associações, composições e agregações). Para tornar isso possível, faremos um formulário com abas, sendo que cada aba conterá um grupo de informações diferentes. Na primeira aba, teremos os dados básicos do cliente, conforme viso na figura a seguir.

The screenshot shows the 'Basic data' tab of a customer form. At the top, there are three tabs: 'Basic data' (selected), 'Skills', and 'Contacts'. Below the tabs are several input fields:

- Code:** An input field with a placeholder.
- Name:** An input field.
- Address:** An input field.
- City:** A dropdown menu with 'Buscar' as the placeholder.
- Phone:** An input field.
- BirthDate:** An input field with a calendar icon.
- Status:** A dropdown menu with 'Selecionar' as the placeholder.
- Email:** An input field.
- Category:** A dropdown menu with 'Selecionar' as the placeholder.
- Gender:** Two radio buttons labeled 'Male' and 'Female'.

At the bottom left are buttons for 'Save', 'Clear', and 'List'. On the right side, it says 'MARCOS ANTONIO RAFAEL DA FONSECA -'.

Figura 117 Aba para dados básicos do cliente

Na segunda aba do formulário, teremos uma lista de habilidades (`skill`), representada por um componente do tipo `TDBCheckGroup`. Nesta aba, poderemos selecionar um ou mais registros. As habilidades (`skill`) não são específicas a um cliente, podendo ser relacionadas em diferentes clientes.

The screenshot shows the 'Skills' tab of the customer form. At the top, there are three tabs: 'Basic data' (selected), 'Skills' (highlighted with a blue border), and 'Contacts'. Below the tabs is a `TDBCheckGroup` component containing a list of skills:

- Leitura
- Escrita
- Comunicação
- Criatividade
- Relações
- Organização
- Liderança

At the bottom left are buttons for 'Save', 'Clear', and 'List'. On the right side, it says 'MARCOS ANTONIO RAFAEL DA FONSECA -'.

Figura 118 Aba com habilidades do cliente

A terceira aba do formulário, apresentará campos para cadastrar contatos. Os campos Type (tipo) e Value (valor) serão campos vetoriais. Assim, o usuário poderá cadastrar várias ocorrências desses campos, por meio do botão “Add”. Estes dados são específicos de um cliente, conforme o conceito de composição.

Figura 119 Aba com contatos do cliente

Para construir este formulário, utilizaremos a classe `BootstrapFormBuilder`, que permite criar um formulário com abas, utilizando a biblioteca Bootstrap.

Faremos o formulário abrir na forma de cortina lateral, o que é realizado pelo método `setTargetContainer()`, que indica o local onde o formulário será aberto.

Conforme pode ser acompanhado no código-fonte a seguir, a construção da página inicia pela criação do formulário (`BootstrapFormBuilder`). Este formulário conterá três abas (Basic data, Contacts, e Skills). Após criarmos o formulário, são criados os vários componentes de entrada de dados no formulário (`TEntry`, `TDBUniqueSearch`, `TCombo`, etc). O campo para busca de cidades `$city_id`, por exemplo, utilizará o componente `TDBUniqueSearch`, que permite localização de registro por digitação.

Em seguida, opções são adicionadas ao botão de rádio para gênero (`gender`) e também a combo de status. Logo após, mais algumas propriedades são definidas.

Em seguida, adicionamos a aba “Basic data” ao formulário, por meio do método `appendPage()`, e os campos são inseridos no formulário por meio do método `addFields()`, que recebe slots de campos representados por vetores, como já visto.

Em seguida, é criada a aba “Skills”, que contém uma lista de habilidades (`$skill_list`), instância da classe `TDBCheckGroup`. Este componente listará no formulário todas as habilidades (`skill`) cadastradas, permitindo uma seleção múltipla.

Logo após, adicionamos a aba “Contacts” que terá um formulário vetorial do tipo `TFieldList`, que permite uma lista repetida de campos. Neste formulário vetorial teremos dois campos: tipo do contato (`contact_type[]`), e valor (`contact_value[]`). Os campos são adicionados ao formulário vetorial por meio do método `addField()`, e o formulário vetorial é adicionado à tela por meio do método `addContent()`.

Neste formulário, são criadas três ações: uma para salvar os dados, conectada ao método `onSave()`; uma para limpar o formulário, conectada ao método `onClear()`; e outra para fechar a cortina lateral, conectada ao método `onClose()`.

app/control/Organization/ComplexViews/CustomerFormView.class.php

```
<?php
class CustomerFormView extends TPage
{
    private $form, $contacts;

    function __construct()
    {
        parent::__construct();
        // habilita apresentação em cortina lateral
        parent::setTargetContainer('adianti_right_panel');

        $this->form = new BootstrapFormBuilder('form_customer');
        $this->form->setFormTitle('Customer');

        // cria os campos do formulário
        $code      = new TEntry('id');
        $name      = new TEntry('name');
        $address   = new TEntry('address');
        $phone     = new TEntry('phone');
        $city_id   = new TDBUniqueSearch('city_id', 'samples', 'City', 'id', 'name');
        $birthdate = new TDate('birthdate');
        $email     = new TEntry('email');
        $gender    = new TRadioGroup('gender');
        $status    = new TCombo('status');
        $category_id = new TDBCombo('category_id', 'samples', 'Category', 'id', 'name');

        // adiciona as opções nas combos
        $gender->addItems( [ 'M' => 'Male', 'F' => 'Female' ] );
        $status->addItems( [ 'S' => 'Single', 'C' => 'Committed', 'M' => 'Married' ] );
        $gender->setLayout('horizontal');

        // define algumas propriedades dos campos
        $code->setEditable(FALSE);
        $code->setSize('30%');
        $city_id->setSize('100%');
        $birthdate->setSize('100%');
        $status->setSize('100%');
        $category_id->setSize('100%');
        $gender->setUseButton();
        $gender->setSize('100%');
        $status->enableSearch();
        $category_id->enableSearch();
        $city_id->setMinLength(0);
        $city_id->setMask('{name} <b>{state->name}</b>');

        // adiciona os campos da primeira aba do formulário
        $this->form->appendPage('Basic data');
        $this->form->addField( [ new TLabel('Code') ], [ $code ] );
        $this->form->addField( [ new TLabel('Name') ], [ $name ] );
        $this->form->addField( [ new TLabel('Address') ], [ $address ] );
        $this->form->addField( [ new TLabel('City') ], [ $city_id ] );
        $this->form->addField( [ new TLabel('Phone') ], [ $phone ],
                               [ new TLabel('BirthDate') ], [ $birthdate ] );
        $this->form->addField( [ new TLabel('Status') ], [ $status ],
                               [ new TLabel('Email') ], [ $email ] );
        $this->form->addField( [ new TLabel('Category') ], [ $category_id ],
                               [ new TLabel('Gender') ], [ $gender ] );

        // adiciona os campos da segunda aba do formulário
        $this->form->appendPage('Skills');
        $skill_list = new TDBCheckGroup('skill_list', 'samples', 'Skill', 'id', 'name');
        $this->form->addField( [ new TLabel('Skill') ], [ $skill_list ] );
    }
}
```

```

// adiciona os campos da terceira aba do formulário
$this->form->appendPage('Contacts');

// cria os campos vetoriais
$contact_type = new TCombo('contact_type[]');
$contact_type->setSize('100%');
$contact_type->addItems( ['email' => 'E-mail', 'phone' => 'Phone' ]);
$contact_value = new TEntry('contact_value[]');
$contact_value->setSize('100%');

// cria o formulário vetorial e adiciona os campos
$this->contacts = new TFieldList;
$this->contacts->addField( '<b>Type</b>', $contact_type, ['width' => '50%']);
$this->contacts->addField( '<b>Value</b>', $contact_value, ['width' => '50%']);
$this->form->addField($contact_type);
$this->form->addField($contact_value);
$this->contacts->enableSorting(); // habilita ordenação

// adiciona formulário vetorial de contatos ao formulário
$this->form->addContent( [ new TLabel('Contacts') ], [ $this->contacts ] );

// adiciona as ações ao formulário
$this->form->addAction( 'Save', new TAction([$this, 'onSave']), 'fa:save' );
$this->form->addActionLink('Clear',new TAction([$this,'onClear']),'fa:eraser');
$this->form->addHeaderActionLink( _t('Close'), new TAction([$this, 'onClose']), 'fa:times red');

parent::add($this->form);
}

```

O método `onSave()` é responsável por armazenar os dados do formulário de clientes e tem algumas características que o diferenciam dos demais métodos de salvamento vistos até aqui. Além de armazenar os dados do cliente como nome, telefone e data de nascimento, ele precisará armazenar a composição com contatos (`Contact`), e a agregação com as habilidades (`Skill`).

Após abrir a transação com a base de dados, o campo de nascimento é validado, e os dados do formulário são obtidos em um Active Record `Customer`, por meio do método `fromArray()`. Em seguida, os registros cadastrados como contatos (`contact_type` e `contact_value`) são percorridos, uma vez que se trata de um campo vetorial que pode possuir muitas ocorrências. Cada um destes registros é adicionado ao cliente por meio do método `addContact()`, já definido no capítulo 3. Quando o cliente for armazenado pelo método `store()`, estes objetos internos (`Contact`) também serão armazenados em sua respectiva tabela.

Após percorrermos os registros de contatos, são percorridos os registros de habilidades (`skill_list`), resultado do componente `TDBCheckGroup`. Este componente também retorna um vetor. Este vetor contém os códigos (`ID`), das habilidades (`Skill`) selecionadas pelo usuário no componente. Para cada código retornado, é instanciado um objeto `Skill`, para que este, seja então adicionado pelo método `addSkill()`.

Por fim, o cliente é armazenado na base de dados por meio do método `store()`. Quando este método é executado, as relações de composição com contatos (`Contact`) e de agregação com habilidades (`Skill`) são tratadas internamente na classe `Customer`, o que já foi implementado no capítulo 3. O código (`id`) gerado é enviado ao formulário

por meio do método `sendData()`, pois como se trata de um método estático, o `onSave()` precisa de alguma maneira enviar o id gerado para o formulário.

A cortina lateral é fechada por meio do método `Template.closeRightPanel()`. Em seguida uma mensagem é exibida ao usuário, e após sua confirmação, adatagrid de clientes é carregada por meio de uma ação (`$posAction`).

```

function onSave()
{
    try {
        // abre transação
        TTransaction::open('samples');

        // valida preenchimento de campo
        if (empty($param['birthdate'])) {
            throw new Exception(...);
        }

        // lê os dados do formulário para preencher um objeto Customer
        $customer = new Customer;
        $customer->fromArray( $param );

        if( !empty($param['contact_type']) AND is_array($param['contact_type']) ) {
            // percorre os dados de contato (type e value) cadastrados
            foreach( $param['contact_type'] as $row => $contact_type ) {
                if ($contact_type) {
                    $contact = new Contact;
                    $contact->type  = $contact_type;
                    $contact->value = $param['contact_value'][$row];

                    // adiciona o contato ao cliente
                    $customer->addContact($contact);
                }
            }
        }

        if ( !empty($param['skill_list']) ) {
            // percorre as habilidades selecionadas
            foreach( $param['skill_list'] as $skill_id ) {
                // adiciona a habilidade ao cliente
                $customer->addSkill(new Skill($skill_id));
            }
        }

        // armazena o objeto cliente
        $customer->store();

        // envia o id gerado para o formulário
        $data = new stdClass;
        $data->id = $customer->id;
        TForm::sendData('form_customer', $data);

        // fecha cortina lateral
        TScript::create("Template.closeRightPanel()");
    }

    $posAction = new TAction(array('CustomerDataGridView', 'onReload'));
    $posAction->setParameter('target_container', 'adanti_div_content');

    // Exibe mensagem de confirmação com ação posterior
    new TMessage('info', 'Registro salvo', $posAction);

    TTransaction::close(); // fecha transação
}

```

```

    catch (Exception $e)
    {
        new TMessage('error', $e->getMessage());
        TTransaction::rollback();
    }
}

```

Assim como o método `onSave()` teve características específicas em função dos relacionamentos de composição e agregação da classe `Customer`, também o método `onEdit()` terá uma implementação diferente. No método `onEdit()`, os cuidados devem ser no momento de carregar as informações para preencher o formulário.

Logo após abrir a transação com a base de dados, carregamos o cliente (`Customer`), conforme a chave do registro '`id`', presente nos parâmetros. Em seguida, carregamos os contatos por meio do método `getContacts()`. Cada contato será adicionado ao formulário vetorial de contatos por meio do método `addDetail()`. Além de adicionar os contatos, também criamos um botão de adicionar “+”, responsável por clonar a última linha do formulário vetorial, por meio do método `addCloneAction()`.

Logo em seguida, são lidas as habilidades do cliente por meio do método `getSkills()`, que retorna uma coleção de objetos do tipo `Skill`. Percorremos estes objetos para coletar os seus códigos (`id`), uma vez que o componente em tela `skill_list`, que é do tipo `TDBChekGroup`, tem como índice o `id` do objeto. Após carregarmos todos os dados, o formulário é preenchido pelo método `setData()`.

```

function onEdit($param)
{
    try {
        if (isset($param['id'])) {
            // abre transação com a base de dados
            TTransaction::open('samples');

            // carrega o cliente
            $customer = new Customer($param['id']);

            // carrega os contatos
            $contacts = $customer->getContacts();

            if ($contacts) {
                // adiciona linha de título do formulário vetorial
                $this->contacts->addHeader();

                // percorre os contatos
                foreach ($contacts as $contact)
                {
                    $contact_detail = new stdClass;
                    $contact_detail->contact_type = $contact->type;
                    $contact_detail->contact_value = $contact->value;

                    // adiciona o contato ao formulário vetorial de contatos
                    $this->contacts->addDetail($contact_detail);
                }
                // adiciona botão de adicionar linha ao final
                $this->contacts->addCloneAction();
            }
            else {
                // se não existirem contatos, adiciona linha em branco
                $this->onClear($param);
            }
        }
    }
}

```

```

// carrega as habilidades
$skills = $customer->getSkills();
$skill_list = array();
if ($skills)
{
    foreach ($skills as $skill)
    {
        // coleta os id's das habilidades
        $skill_list[] = $skill->id;
    }
}
// preenche o checkbox com um array de id's
$customer->skill_list = $skill_list;

// preenche o formulário com base no objeto
$this->form->setData($customer);

TTransaction::close(); // fecha a transação
}
else
{
    $this->onClear($param);
}
}
catch (Exception $e)
{
    // exibe mensagem de exceção e desfaz operações
    new TMessage('error', $e->getMessage());
    TTransaction::rollback();
}
}

```

Para finalizar, o método `onClear()` será executado sempre que, no momento da edição, for detectado que não existem contatos para serem editados. Neste caso, é criada uma linha com um contato vazio, por meio do método `addDetail()`, passando um objeto vazio como parâmetro.

```

public function onClear($param)
{
    $this->form->clear();

    $this->contacts->addHeader();
    $this->contacts->addDetail( new stdClass );
    $this->contacts->addCloneAction();
}

```

O método `onClose()` é chamado a partir de um botão definido no método construtor e exibido no topo do formulário. Seu objetivo é fechar a cortina lateral.

```

public static function onClose($param)
{
    TScript::create("Template.closeRightPanel()");
}

```

Obs: O método `onClear()` deve ser executado sempre que o usuário acessar o formulário de cadastro vazio. Assim, este método já deixará uma linha de contato vazia para ser preenchida. Portanto, o método `onClear()` deve ser chamado a partir do menu.

5.3.2 Listagem de clientes

Neste exemplo, criaremos umadatagrid de clientes com autofiltro no cabeçalho das colunas. Assim, cada coluna terá um input superior no qual o usuário poderá digitar conteúdo para filtrar os dados daquela coluna. Além desses filtros, adatagrid terá botões na parte superior para exportar os dados em diferentes formatos (CSV, PDF, XML), bem como um atalho (Novo) para o formulário de cadastro de clientes. Adatagrid também terá os tradicionais botões de edição e exclusão, bem como uma paginação de registros. Na figura a seguir, conferimos o resultado deste exemplo.

Lista de clientes					
MARCOS ANTONIO RAFAEL DA FONSECA -					
<input checked="" type="checkbox"/>	<input type="button" value="Delete"/>	1	Andrei Zmievski	Rua Palo Alto	Caxias do Sul (RS)
<input checked="" type="checkbox"/>	<input type="button" value="Delete"/>	2	Rubens Prates	Rua Campinas, 123	São Paulo (SP)
<input checked="" type="checkbox"/>	<input type="button" value="Delete"/>	3	Augusto Campos	Rua BRLinux, 343	São Paulo (SP)
<input checked="" type="checkbox"/>	<input type="button" value="Delete"/>	4	Marcelio Leal	Rua Belém, 334	São Paulo (SP)
<input checked="" type="checkbox"/>	<input type="button" value="Delete"/>	5	Manuel Lemos	Rua Osasco, 949	São Paulo (SP)
<input checked="" type="checkbox"/>	<input type="button" value="Delete"/>	6	Fábio Locatelli	Rua Lachada, 012	Lajeado (RS)
<input checked="" type="checkbox"/>	<input type="button" value="Delete"/>	7	Leonardo Soldatelli	Rua Tramandaí, 234	Porto Alegre (RS)
<input checked="" type="checkbox"/>	<input type="button" value="Delete"/>	8	Alberto Bengoa	Rua Porto, 23	Porto Alegre (RS)
<input checked="" type="checkbox"/>	<input type="button" value="Delete"/>	9	Fábio Milani	Rua Canela, 34	Caxias do Sul (RS)
<input checked="" type="checkbox"/>	<input type="button" value="Delete"/>	10	Huberto Meyer	Rua Orquestra, 101	Porto Alegre (RS)

Figura 120 Datagrid de clientes

Nesta página, utilizaremos os métodos padronizados dedatagrid fornecidos pelo framework e importados pelo Trait `AdiantiStandardListTrait`, tais como `onSearch()` e `onReload()` para filtro e carregamento, `onDelete()` e `Delete()` para exclusão de registros.

Conforme pode ser visto no código-fonte a seguir, iniciamos a criação da página com uma série de definições, tais como: a conexão de base de dados, por meio do método `setDatabase()`; a classe a ser manipulada, por meio do método `setActiveRecord()`; a ordenação padrão, por meio do método `setDefaultOrder()`. Em seguida são definidos os filtros do formulário, por meio do método `addFilterField()`. É importante notar que este método recebe como parâmetros: o nome do campo da tabela, o operador e o nome do campo do formulário, estabelecendo uma relação entre ambos. Na última chamada do método `addFilterField()`, definimos o filtro por nome de cidade. Mas, como na tabela de clientes não existe campo com o nome da cidade (somente com seu código), precisamos escrever uma subconsulta SQL para buscar o nome da cidade, para então utilizá-lo no filtro de buscas, comparando-o por meio do operador like, com o campo `city_name`, que é o input que estará no topo da coluna de filtro da cidade.

O método `setOrderCommand()` é utilizado neste caso por que uma das colunas dadatagrid é formada por uma coluna associada (`city->name`). Como esta coluna não existe e na verdade é só um atalho para chamada de um método correspondente, quando o usuário clicar no cabeçalho da coluna para ordenar, o Framework precisa saber o correto comando a ser executado para ordenação (`order by`). Para tal, o método `setOrderCommand()` está dizendo para o Framework utilizar o comando SQL do segundo argumento, sempre que o usuário clicar na coluna identificada pelo primeiro argumento.

Após várias definições, é criado um formulário (`TForm`). Este formulário englobará toda adatagrid, mas não será um objeto visível. Ele somente é necessário pois teremos campos de busca (inputs do cabeçalho), que dispararão ações usando postagem de dados.

Em seguida, são criados campos para filtro de dados (`$id`, `$name`, `$address`, `$city_name`, e `$gender`). Após, habilitamos o método `exitOnEnter()` para disparar o método de saída de campo (exit action) sempre que o usuário pressionar a tecla ENTER. Em seguida os tamanhos dos campos são ajustados pelo método `setSize()`, e desligamos o foco dos campos, definindo o `tabindex` de cada campo. É importante desligarmos o foco para que na saída de um campo, o usuário não caia no próximo, disparando mais um filtro.

Em seguida, definimos as ações de saída de cada um dos campos de busca. Todos os campos de busca estarão vinculados ao método `onSearch()` que é um método padrão fornecido pelo Trait `AdiantiStandardListTrait`. Este método já foi visto anteriormente, e é responsável por verificar o que o usuário preencheu na busca, e criar um filtro em sessão posteriormente utilizado pelo método `onReload()`.

Após definir as ações de saída dos campos de busca, é criada adatagrid (`TDataGrid`), definida sua largura, e habilitado um popover sobre suas linhas. Em seguida, são criadas as colunas (`TGridColumn`). No caso da coluna com o nome da cidade, utilizamos a sintaxe `{city->name}`, que acaba disparando indiretamente o método `getCity()`, que retornará primeiro o objeto `City`, para então obter o seu nome. Para cada coluna, é definida uma ação de ordenação (`setAction`), vinculada ao método `onReload()`, sendo que para cada chamada, é definida uma ordenação diferente (parâmetro `order`). Sobre a coluna de gênero é ainda definido um transformer, para exibir o nome do gênero por extenso. Por fim, as colunas são adicionadas àdatagrid pelo método `addColumn()`.

Estadatagrid terá duas ações de linha, que serão: Editar, vinculada ao formulário `CustomerFormView`; e excluir, vinculada ao método `onDelete()` fornecido pelo Trait `AdiantiStandardListTrait`, que já possui a implementação padrão para exclusão.

Após executarmos o método `createModel()`, que cria a estrutura dadatagrid em memória, adatagrid é adicionada ao formulário. Em seguida, criamos uma linha (`$tr`), que conterá os campos de filtro (`$id`, `$name`, `$address`, `$city_name`, e `$gender`). Esta linha é adicionada ao início dadatagrid por meio do método `prependRow()`.

Em seguida, os campos são adicionados logicamente ao formulário por meio do método `addField()`. O método `setData()` é usado para manter o formulário preenchido com os últimos filtros aplicados, mesmo que o usuário recarregue a página.

Por fim, é criado o objeto de paginação (`TPageNavigation`), e um painel (`TPanelGroup`) que conterá o formulário e a paginação. Por fim, criamos um objeto de menu flutuante (`TDropDown`), que conterá as ações de exportação da datagrid. As ações estarão conectadas aos métodos: `onExportCSV()`, para exportar em CSV, `onExportPDF()`, para exportar em PDF, e `onExportXML()`, para exportar em XML. Haverá ainda um botão “Novo” conectado ao formulário `CustomerFormView`, para iniciar um novo cadastro de clientes. Ambos dropdown e o botão de novo são adicionados na parte superior direita do painel, o que se dá pela chamada dos métodos `addHeaderWidget()`, para adicionar o dropdown, e `addHeaderActionLink()` para adicionar um atalho para o formulário de edição.

app/control/Organization/ComplexViews/CustomerDataGridView.class.php

```
<?php
class CustomerDataGridView extends TPage
{
    private $form;           // formulário lógico de buscas
    private $datagrid;        // datagrid
    private $pageNavigation;

    // importa ações padronizada para datagrid
    use Adianti\Base\AdiantiStandardListTrait;

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

        $this->setDatabase('samples'); // conexão de base de dados
        $this->setActiveRecord('Customer'); // classe manipulada
        $this->setDefaultOrder('id', 'asc'); // ordenação padrão
        $this->addFilterField('id', '=', 'id'); // define filtro
        $this->addFilterField('name', 'like', 'name'); // define filtro
        $this->addFilterField('address', 'like', 'address'); // define filtro
        $this->addFilterField('gender', '=', 'gender'); // define filtro

        // define filtro usando subconsulta
        $this->addFilterField('($SELECT name from city WHERE id=customer.city_id)', 'like', 'city_name');

        // define SQL customizado para ordenação no nome da cidade
        $this->setOrderCommand('city->name', '(select name from city where city_id = id)');

        // cria o formulário lógico
        $this->form = new TForm('form_search_customer');

        // cria os campos de busca
        $id      = new TEntry('id');
        $name   = new TEntry('name');
        $address = new TEntry('address');
        $city_name = new TEntry('city_name');
        $gender   = new TCombo('gender');

        // adiciona os items na combo de gênero
        $gender->addItems( [ 'M' => 'Male', 'F' => 'Female' ] );
    }
}
```

```

// habilita saida de campo ao ENTER
$id->exitOnEnter();
$name->exitOnEnter();
$address->exitOnEnter();
$city_name->exitOnEnter();

// define tamanho dos campos
$id->setSize('100%');
$name->setSize('100%');
$address->setSize('100%');
$city_name->setSize('100%');
$gender->setSize('70');

// evita que o campo receba foco no tab
$id->tabindex = -1;
$name->tabindex = -1;
$address->tabindex = -1;
$city_name->tabindex = -1;
$gender->tabindex = -1;

// define ação de saída dos campos para busca de registros
$id->setExitAction( new TAction([$this, 'onSearch'], ['static'=>'1']) );
$name->setExitAction( new TAction([$this, 'onSearch'], ['static'=>'1']) );
$address->setExitAction( new TAction([$this, 'onSearch'], ['static'=>'1']) );
$city_name->setExitAction( new TAction([$this, 'onSearch'], ['static'=>'1']) );
$gender->setChangeAction( new TAction([$this, 'onSearch'], ['static'=>'1']) );

// cria adatagrid
$this->datagrid = new BootstrapDatagridWrapper(new TDataGrid);
$this->datagrid->style = 'width: 100%';
$this->datagrid->enablePopover('Popover', 'Hi <b>{name}</b>, <br> that lives at
<b>{city->name} - {city->state->name}</b>');

// cria as colunas dadatagrid
$col_id      = new TDataGridColumn('id', 'Id', 'center', '10%');
$col_name    = new TDataGridColumn('name', 'Name', 'left', '28%');
$col_address = new TDataGridColumn('address', 'Address', 'left', '28%');
$col_city    = new TDataGridColumn('{city->name} ({city->state->name})', 'City',
'left', '28%');
$col_gender  = new TDataGridColumn('gender', 'Gender', 'left', '6%');

// define ação de ordenação para as colunas
$col_id->setAction(new TAction([$this, 'onReload']), ['order' => 'id']);
$col_address->setAction(new TAction([$this, 'onReload']), ['order' =>
'address']);
$col_name->setAction(new TAction([$this, 'onReload']), ['order' => 'name']);
$col_city->setAction(new TAction([$this, 'onReload']), ['order' => 'city-
>name']);

// transforma o gênero para exibir por extenso
$col_gender->setTransformer( function ($value) {
    return $value == 'F' ? 'Female' : 'Male';
});

// adiciona as colunas àdatagrid
$this->datagrid->addColumn($col_id);
$this->datagrid->addColumn($col_name);
$this->datagrid->addColumn($col_address);
$this->datagrid->addColumn($col_city);
$this->datagrid->addColumn($col_gender);

// cria duas ações de linha (editar e excluir)
$action1 = new TDataGridAction(['CustomerFormView', 'onEdit'], ['id'=>'{id}' ,
'register_state' => 'false']);
$action2 = new TDataGridAction([$this, 'onDelete'], ['id'=>'{id}']);

```

```

// adiciona as ações à datagrid
$this->datagrid->addAction($action1, 'Edit', 'fa:edit blue');
$this->datagrid->addAction($action2, 'Delete', 'fa:trash red');

// cria a estrutura dadatagrid em memória
$this->datagrid->createModel();

// adicionadatagrid ao formulário
$this->form->add($this->datagrid);
$this->form->style = 'overflow-x:auto';

// cria linha com os inputs de busca
$tr = new TElement('tr');
$this->datagrid->prependRow($tr);

$tr->add( TElement::tag('td', ''));
$tr->add( TElement::tag('td', ''));
$tr->add( TElement::tag('td', $id));
$tr->add( TElement::tag('td', $name));
$tr->add( TElement::tag('td', $address));
$tr->add( TElement::tag('td', $city_name));
$tr->add( TElement::tag('td', $gender));

$this->form->addField($id);
$this->form->addField($name);
$this->form->addField($address);
$this->form->addField($city_name);
$this->form->addField($gender);

// mantém o formulário preenchido com os últimos dados de busca
$this->form->setData( TSession::getValue(__CLASS__.'_filter_data'));

// cria o paginador
$pageNavigation = new TPageNavigation();
$pageNavigation->setAction(new TAction([$this, 'onReload']));
$pageNavigation->setWidth($this->datagrid->getWidth());
$pageNavigation->enableCounters();

// encapsula o formulário e paginador em um painel
$panel = new TPanelGroup(_t('Customer list'));
$panel->add($this->form);
$panel->addFooter($this->pageNavigation);

// cria um menu flutuante com opções de exportação
$dropdown = new TDropDown('Export', 'fa:list');
$dropdown->setButtonClass('btn btn-default waves-effect dropdown-toggle');
$dropdown->addAction('Save as CSV', new TAction([$this, 'onExportCSV']),
    ['register_state' => 'false', 'static'=>'1'], 'fa:table blue' );
$dropdown->addAction('Save as PDF', new TAction([$this, 'onExportPDF']),
    ['register_state' => 'false', 'static'=>'1'], 'fa:file-pdf-o red' );
$dropdown->addAction('Save as XML', new TAction([$this, 'onExportXML']),
    ['register_state' => 'false', 'static'=>'1'], 'fa:code green' );
$panel->addHeaderWidget( $dropdown );

$panel->addHeaderActionLink( 'New', new TAction(['CustomerFormView', 'onEdit'],
    ['register_state' => 'false']), 'fa:plus green' );

// cria uma caixa vertical para empacotar o breadcrumb e painel.
$vbox = new TVBox;
$vbox->style = 'width: 100%';
$vbox->add(new TXMLBreadCrumb('menu.xml', __CLASS__));
$vbox->add($panel);

parent::add($vbox);
}

}

```

5.3.3 Formulário de Produtos

O objetivo deste exemplo é demonstrar como lidar com campos de upload de arquivos. Para tal, implementaremos um cadastro de produtos com dois campos para upload de arquivos: um simples (`TFile`), e um de múltiplos arquivos (`TMultiFile`).

Na imagem a seguir, temos a tela do cadastro de produtos com campos para Id, descrição, estoque, preço de venda, unidade, foto do produto, e demais imagens.

The screenshot shows a web-based form for product registration. At the top left, it says "Produto". Below that, there's a note: "MARCOS ANTONIO RAFAEL DA FONSECA -". The form fields include:

- ID:** A text input field containing the value "1".
- Description:** A text input field containing the value "Pendrive 512Mb".
- Stock:** A text input field containing "10,00".
- Sale Price:** A text input field containing "57,60".
- Unity:** A dropdown menu currently set to "Pieces".
- Photo Path:** A file input field with the placeholder "Escolher arquivo" and the file path "files/images/1/pendrive.jpg".
- Images:** A file input field with the placeholder "Escolher arquivos" and the message "Nenhum arquivo selecionado".
- Images Preview:** Two thumbnail images of the pendrive are shown below the input fields.

At the bottom left of the form are three buttons: "Save", "Clear", and "List". At the bottom right, it says "MARCOS ANTONIO RAFAEL DA FONSECA --".

Figura 121 Formulário de cadastro de produtos

Como manipulação de arquivos envolve operações como copiar arquivos, definir um nome único, e atualizar o registro na base de dados, vamos utilizar métodos pré-definidos pelo Framework para estas operações. Estes métodos estão no Trait `AdiantiFileSaveTrait`, que fornece alguns métodos úteis para copiar arquivos, utilizados em seguida.

O cadastro de produtos possui um campo que é a foto do produto (`photo_path`). Para este campo, utilizaremos um componente de upload simples (`TFile`). O cadastro terá também o campo `images`, que conterá uma lista de imagens. Para este campo, utilizaremos o componente para upload múltiplo (`TMultiFile`). Estas imagens são armazenadas em uma tabela relacionada (`product_image`).

O programa inicia com a importação do Trait `AdiantiFileSaveTrait`, que fornecerá os métodos de manipulação de arquivos. No método construtor, começamos com a criação do formulário `BootstrapFormBuilder`. Em seguida criamos os campos do formulário. Para ambos componentes de upload de arquivos (`TFile`, e `TMultiFile`), as extensões permitidas são restritas (`gif`, `png`, `jpg`, `jpeg`) por meio do método `setAllowedExtensions()`. O método `enableFileHandling()` habilita tratamento avançado de arquivos, com barra de progresso, e botão de exclusão. O método `enablePopover()` habilita preview da imagem com popover. Já o método

`enableImageGallery()` habilita galeria de imagem, com arquivos exibidos na forma de blocos lado a lado.

Em seguida, os campos são adicionados ao formulário por meio `addFields()`, e algumas validações são programadas por meio do método `addValidation()`. O formulário terá algumas ações que são: Salvar, vinculada ao método `onSave()`, limpar, vinculada ao método `onEdit()`, e Listar, vinculada à `ProductList`.

app/control/Organization/ComplexViews/ProductForm.class.php

```
<?php
class ProductForm extends TPage
{
    protected $form;

    // importa operações padrão para manipulação de arquivos
    use Adianti\Base\AdiantiFileSaveTrait;

    function __construct()
    {
        parent::__construct();
        $this->form = new BootstrapFormBuilder('form_Product');
        $this->form->setFormTitle(_t('Product'));

        // cria os campos do formulário
        $id          = new TEntry('id');
        $description = new TEntry('description');
        $stock       = new TEntry('stock');
        $sale_price  = new TEntry('sale_price');
        $unity       = new TCombo('unity');
        $photo_path  = new TFile('photo_path');
        $images      = new TMultiFile('images');

        // limita extensões permitidas
        $photo_path->setAllowedExtensions( ['gif', 'png', 'jpg', 'jpeg'] );
        $images->setAllowedExtensions( ['gif', 'png', 'jpg', 'jpeg'] );

        // habilita barra de progresso, preview
        $photo_path->enableFileHandling();
        $photo_path->enablePopover();

        // habilita barra de progress, preview, modo galeria
        $images->enableFileHandling();
        $images->enableImageGallery();
        $images->enablePopover('Preview', '');

        $id->setEditable( FALSE );
        $entity->addItems( ['PC' => 'Pieces', 'GR' => 'Grain'] );
        $stock->setNumericMask(2, ',', '.', TRUE); // TRUE: process mask when editing
and saving
        $sale_price->setNumericMask(2, ',', '.', TRUE); // TRUE: process mask when
editing and saving

        // adiciona campos ao formulário
        $this->form->addField( [new TLabel('ID', 'red')], [$id] );
        $this->form->addField( [new TLabel('Description', 'red')], [$description] );
        $this->form->addField( [new TLabel('Stock', 'red')], [$stock],
[new TLabel('Sale Price', 'red')], [$sale_price] );
        $this->form->addField( [new TLabel('Unity', 'red')], [$unity] );
        $this->form->addField( [new TLabel('Photo Path')], [$photo_path] );
        $this->form->addField( [new TLabel('Images')], [$images] );
        $id->setSize('50%');
```

```

    // adiciona validações
    $description->addValidation('Description', new TRequiredValidator);
    $stock->addValidation('Stock', new TRequiredValidator);
    $sale_price->addValidation('Sale Price', new TRequiredValidator);
    $unity->addValidation('Unity', new TRequiredValidator);

    // adiciona ações
    $this->form->addAction( 'Save', new TAction([$this, 'onSave']), 'fa:save green');
    $this->form->addActionLink( 'Clear', new TAction([$this, 'onEdit']), 'fa:eraser red');
    $this->form->addActionLink( 'List', new TAction(['ProductList', 'onReload']), 'fa:table blue');

    parent::add($this->form);
}

```

O método `onSave()` será implementado para não somente gravar o registro de produto na base de dados, mas também para copiar a imagem do produto para o seu respectivo diretório. Ele inicia com a abertura da transação com a base de dados. Logo em seguida, roda as validações de formulário, obtém os dados com o método `getData()`, e grava o objeto no banco pelo método `store()`. Em seguida, é utilizado o método `saveFile()` importado do Trait `AdiantiFileSaveTrait` para copiar o arquivo identificado pelo objeto `photo_path` do formulário, para a pasta `files/images`. Já o método `saveFiles()` é utilizado para salvar as várias imagens do campo `images` para a pasta `files/images`. As imagens do campo `images`, são armazenadas na tabela `ProductImage`, no atributo `image`, relacionada à tabela principal pelo atributo `product_id`.

```

public function onSave()
{
    try {
        TTransaction::open('samples');

        $this->form->validate(); // roda validações
        $data = $this->form->getData(); // obtém dados do form

        // grava o produto
        $object = new Product;
        $object->fromArray( (array) $data );
        $object->store();

        // copia o arquivo para a pasta de destino, e atualiza o objeto
        $this->saveFile($object, $data, 'photo_path', 'files/images');

        // copia os arquivos para a pasta de destino, e atualiza tabelas
        $this->saveFiles($object, $data, 'images', 'files/images', 'ProductImage',
        'image', 'product_id');

        // manda os dados para o form
        $data->id = $object->id;
        $this->form->setData($data);

        TTransaction::close();
        new TMessage('info', AdiantiCoreTranslator::translate('Record saved'));
    }
    catch (Exception $e) {
        $this->form->setData($this->form->getData());
        new TMessage('error', $e->getMessage());
        TTransaction::rollback();
    }
}

```

Os métodos `saveFile()` e `saveFiles()` copiam os arquivos para a `files/images` e também criam uma estrutura de subdiretório com o `id` do registro do banco, a fim de evitar conflitos de nomes. Ao final do procedimento de cópia dos arquivos, estes métodos também atualizam o registro na base de dados com o nome final gerado.

Assim ao salvar precisamos “tratar” os arquivos, o mesmo ocorre em sua edição, com o carregamento dos arquivos. Neste caso, além de carregar o registro do produto em si, precisamos carregar as imagens correspondentes. Como a foto do produto (`photo_path`), é armazenada em um campo da própria tabela, quanto à esse campo nada precisamos fazer, pois ele já é carregado automaticamente. Já quanto ao campo de múltiplas imagens (`images`), precisamos carregar estas imagens a partir da tabela relacionada (`ProductImage`), com operações de `where()` e `getIndexedArray()`, que retorna um vetor indexados com todas as imagens encontradas para aquele id de produto. Com a lista de imagens carregada em memória, esta informação é automaticamente passada para o componente, pois ele receberá os dados por meio do método `setData()`.

```
public function onEdit($param)
{
    try
    {
        if (isset($param['key']))
        {
            // abre transação
            TTransaction::open('samples');

            // carrega o produto
            $object = new Product( $param['key'] );

            // carrega as imagens a partir da tabela relacionada
            $object->images = ProductImage::where('product_id','=', $param['key'])
                ->getIndexedArray('image');

            // envia informações para os componentes do formulário
            $this->form->setData($object);

            // fecha transação
            TTransaction::close();
            return $object;
        }
        else
        {
            $this->form->clear();
        }
    }
    catch (Exception $e) // in case of exception
    {
        new TMessage('error', $e->getMessage());
        TTransaction::rollback();
    }
}
```

5.3.4 Listagem de Produtos

Neste exemplo, construiremos uma listagem de produtos, utilizando os métodos padrão fornecidos pelo Framework (Trait `AdiantiStandardListTrait`) para operações de busca, ordenação e paginação. Na figura a seguir, podemos conferir adatagrid de produtos, que terá dois campos de busca: descrição e unidade, bem como várias colunas exibidas: Id, Descrição, Estoque, Preço, e Unidade. Ainda teremos um preview da imagem quando passarmos o mouse sobre o registro dadatagrid.

ID	Description	Stock	Sale Price	Unit
1	Pendrive 512Mb	10.0	57.6	PC
2	HD 120 GB	20.0	180.0	
3	SD CARD 512MB	4.0	35.0	
4	SD CARD 1GB MINI	3.0	40.0	
5	CAM. PHOTO I70 Silver	5.0	900.0	PC
6	CAM. PHOTO DSC-W50 Silver	4.0	700.0	PC
7	WEBCAM INSTANT VF0040SP	4.0	80.0	PC
8	CPU 775 CEL.D 360 3.46 533M	10.0	300.0	PC
9	Recorder DCR-DVD108	2.0	1400.0	PC
10	HD IDE 80G 7.200	8.0	160.0	PC

1 a 10 de 24 registros

MARCOS ANTONIO RAFAEL DA FONSECA -

Figura 122 Listagem de produtos

Adatagrid de produtos é toda definida em seu método construtor, onde executamos alguns métodos importados do Trait `AdiantiStandardListTrait` como `setDatabase()`, para definir a base de dados, `setActiveRecord()`, para definir a classe a ser manipulada; `setDefaultOrder()`, para definir a ordenação padrão; e, `addFilterField()`, para definir quais serão os campos utilizados para a busca, bem como quais serão os operadores utilizados para a comparação.

Após as definições básicas, criamos um formulário (`BootstrapFormBuilder`), e criamos os campos de busca (`description`, e `unit`). Em seguida, adicionamos os campos ao formulário (`addFields`), e definimos as ações (`addAction`), vinculando as ações a métodos importados do Trait, como é o caso do método `onSearch()`, para buscas.

Neste exemplo, também utilizamos o método `setData()`, para manter o formulário preenchido com os dados de busca da seção. A variável de seção `<classe>_filter_data` contém os dados de busca do formulário. Este nome de variável é padronizada, sempre que utilizamos o Trait `AdiantiStandardListTrait`.

Em seguida, criamos a datagrid (`TDataGrid`), bem como as colunas da datagrid (`TGridColumn`), e adicionamos as colunas à datagrid pelo método `addColumn()`. Para que tenhamos preview da imagem ao passar o mouse sobre o registro, utilizamos o método `enablePopover()`, passando o caminho da imagem `{photo_path}`.

Adatagrid terá duas ações, sendo a primeira para edição, vinculada ao método `onEdit()` da `ProductForm`, e a segunda para exclusão, vinculada ao método `onDelete()` do Trait `AdiantiStandardListTrait`. As ações são adicionadas pelo método `addAction()`.

Por fim, criamos a paginação (`TPageNavigation`), e um container (`TVBox`) para empacotar os vários objetos criados. Dentro da caixa vertical, teremos o breadcrumb, o formulário de buscas, e um painel com adatagrid e o paginador.

app/control/Organization/ComplexViews/ProductList.class.php

```
<?php
class ProductList extends TPage
{
    protected $form;      // formulário
    protected $datagrid;  // datagrid
    protected $pageNavigation;

    // trait com onReload, onSearch, onDelete...
    use Adianti\Base\AdiantiStandardListTrait;

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

        $this->setDatabase('samples');                      // define o banco
        $this-> setActiveRecord('Product');                // define a classe
        $this-> setDefaultOrder('id', 'asc');               // define a ordem
        $this-> addFilterField('description', 'like');     // adiciona filtro
        $this-> addFilterField('unity', '=');               // adiciona filtro

        // cria o formulário
        $this-> form = new BootstrapFormBuilder('form_search_Product');
        $this-> form-> setFormTitle(_t('Product list'));

        // cria os campos do formulário
        $description = new TEntry('description');
        $unit        = new TCombo('unity');
        $unit-> addItems( ['PC' => 'Pieces', 'GR' => 'Grain'] );

        // adiciona os campos ao formulário
        $this-> form-> addFields( [new TLabel('Description')], [$description] );
        $this-> form-> addFields( [new TLabel('Unit')], [$unit] );
    }
}
```

```

// mantém o formulário com os dados de busca
$this->form->setData( TSession::getValue('ProductList_filter_data') );

// adiciona as ações ao formulário
$this->form->addAction( 'Find', new TAction([$this, 'onSearch']), 'fa:search');
$this->form->addActionLink( 'New', new TAction(['ProductForm', 'onEdit']),
'fa:plus green');

// cria adatagrid
$this->datagrid = new BootstrapDatagridWrapper(new TDataGrid);
$this->datagrid->enablePopover('Image', '<img style="max-height: 300px'
src={photo_path}>');

// cria as colunas dadatagrid
$col_id          = new TDataGridColumn('id', 'ID', 'center', '10%');
$col_description = new TDataGridColumn('description', 'Description', 'left',
'45%');
$col_stock       = new TDataGridColumn('stock', 'Stock', 'right', '15%');
$col_sale_price  = new TDataGridColumn('sale_price', 'Sale Price', 'right',
'15%');
$col_unity        = new TDataGridColumn('unity', 'Unit', 'right', '15%');

// adiciona as colunas nadatagrid
$this->datagrid->addColumn($col_id);
$this->datagrid->addColumn($col_description);
$this->datagrid->addColumn($col_stock);
$this->datagrid->addColumn($col_sale_price);
$this->datagrid->addColumn($col_unity);

// cria ações de linha (editar e excluir)
$action1 = new TDataGridAction(['ProductForm', 'onEdit'], ['id=>'{id}']);
$action2 = new TDataGridAction([$this, 'onDelete'], ['id=>'{id}']);

// adiciona ações nadatagrid
$this->datagrid->addAction($action1, 'Edit', 'fa:edit blue');
$this->datagrid->addAction($action2, 'Delete', 'fa:trash red');

// cria a estrutura dadatagrid em memória
$this->datagrid->createModel();

// cria o paginador
$pageNavigation = new TPageNavigation();
$pageNavigation->enableCounters();
$pageNavigation->setAction(new TAction([$this, 'onReload']));
$pageNavigation->setWidth($this->datagrid->getWidth());

// cria um container (caixa vertical) para empacotar os objetos
$container = new TVBox();
$container->style = 'width: 100%';
$container->add($this->form);
$container->add(TPanelGroup::pack('', $this->datagrid, $this->pageNavigation));

// adiciona o container na página
parent::add($container);
}
}

```

5.3.5 Formulário mestre-detalhe de vendas

Nos exemplos anteriores, criamos variados tipos de cadastros, mas sempre envolvendo uma única tabela. Neste exemplo, criaremos um formulário para cadastro de vendas, onde teremos um formulário mestre-detalhe, onde o registro mestre terá os dados da venda, e os registros dos detalhes terão os itens da venda, relacionados à produtos.

A figura a seguir demonstra o formulário que criaremos. Na parte superior temos atributos relativos à venda, que é o registro principal. Nesta parte temos atributos como o id, data, cliente, e observação da venda. Já na parte inferior, temos os detalhes da venda. Os detalhes da venda são formados por um formulário de cadastro e edição de itens, bem como de uma datagrid que permite editar ou excluir o registro do item. Os itens possuem informações como o produto, preço, quantidade, e desconto. Ao clicar no botão de editar da datagrid, os dados da linha clicada são transportados para o formulário logo acima para permitir a edição do item. O botão registrar transfere os dados editados para a datagrid. O botão Salvar, grava a venda, bem como os itens.

ProID	Product	Amount	Price	Discount	Subtotal
1	Pendrive 512Mb	1.0	R\$ 40,00	R\$ 0,00	R\$ 40,00
2	HD 120 GB	2.0	R\$ 180,00	R\$ 0,00	R\$ 360,00
3	SD CARD 512MB	3.0	R\$ 35,00	R\$ 0,00	R\$ 105,00

Figura 123 Formulário mestre-detalhe de vendas

Durante a edição de uma venda, os dados dos registros dos detalhes são armazenados em campos escondidos (hidden) dentro da própria datagrid com os itens. Estes campos escondidos são vetoriais [], o que permite que se armazene vários deles. Cada vez que adicionamos um item no detalhe, estamos adicionando também vários campos escondidos contendo os dados daquele item (produto, preço, desconto, ...).

O código inicia no método construtor onde definimos toda a interface do formulário. Como optamos por criar o formulário em uma janela (`TWindow`), utilizamos métodos como `setSize()` para definir o tamanho (80% de largura), `removePadding()`, para remover o espaço extra entre o conteúdo e as bordas da janela, `removeTitleBar()` para remover a barra de título padrão das janelas, e `disableEscape()` para desabilitar o fechamento da janela quando o usuário pressionar a tecla Escape.

Ainda no método construtor, é criado o formulário (`BootstrapFormBuilder`), e em seguida são criados os objetos do formulário principal, bem como os objetos dos detalhes. Aqui destacamos os campos para busca de cliente e produto, nos quais utilizamos o componente `TDBUniqueSearch`, que permite busca de código baseada na digitação.

Depois de declarar os objetos, definimos algumas propriedades como tamanho, pelo método `setSize()`, e mínimo de caracteres para a busca, com o `setMinLength()`. Após, adicionamos algumas validações com o `addValidation()`, e uma ação a ser executada sempre que o usuário alterar o produto, pelo método `setChangeAction()`. Neste caso, sempre que o produto for escolhido, o método `onProductChange()` será executado. Neste caso, ele deverá autocompletar o campo do preço do produto.

Os campos são adicionados ao formulário pelo método `addField()`. Logo após adicionar os campos do registro mestre, adicionamos uma divisória com o método `addContent()`, e em seguida os campos dos detalhes novamente com o `addField()`.

Após adicionar os campos do registro mestre e dos detalhes, criamos a datagrid que exibirá os registros de detalhe (`TDataGrid`). O método `makeScrollable()` juntamente com o método `setHeight()` liga a rolagem vertical dadatagrid. O método `setId()` define um identificador único para ela, e o método `generateHiddenFields()` gera campos escondidos (hidden) para cada coluna dadatagrid. Este método é de suma importância, pois sem ele, não conseguiremos enviar os dados dos detalhes na postagem no momento de salvar a venda. Em seguida, criamos as colunas (`TGridColumn`) destadatagrid, e adicionamos elas com o método `addColumn()`.

Utilizamos o método `setTransformer()` sobre a uma determinada coluna (`$col_descr`) que inicialmente foi criada para exibir o id do produto. Este transformer retorna o nome com base neste ID, possibilitando exibir a sua descrição. A coluna `$col_id` armazena o ID do detalhe, e a coluna `$col_uniq`, armazena um ID aleatório e único. Estas colunas são usadas para controle interno, e não precisam ser exibidas ao usuário. Para tal, é utilizado o método `setVisibility(false)`.

Estadatagrid terá duas ações, que permitirão: editar o item de detalhe, por meio do método `onEditItemProduto()`, e excluir um item do detalhe, pelo método `onDeleteItem()`. Ao utilizarmos '*' no `setFields()`, indicamos que precisaremos de todos atributos.

As colunas preço, desconto, e subtotal terão uma função de transformação que formatará as casas decimais e adicionará a moeda, o que é feito pelo `setTransformer()`.

Adatagrid com os itens será apresentada dentro de um painel (`TPanelGroup`). O formulário terá alguns botões, que são: Close, vinculado ao método `onClose()`, que efetua o fechamento da janela; Save, vinculado ao método `onSave()`, que salvará a venda e os itens; e Clear, vinculado ao método `onClear()`, que limpará o formulário.

app/control/Organization/ComplexViews/SaleForm.class.php

```
<?php
class SaleForm extends TWindow
{
    protected $form;

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

        // define propriedades da janela
        parent::setSize(0.8, null);
        parent::removePadding();
        parent::removeTitleBar();
        parent::disableEscape();

        // cria o formulário
        $this->form = new BootstrapFormBuilder('form_Sale');
        $this->form->setFormTitle('Sale');
        $this->form->setProperty('style', 'margin:0; border:0');

        // cria os campos mestre
        $id          = new TEntry('id');
        $date        = new TDate('date');
        $customer_id = new TDBUniqueSearch('customer_id', 'samples', 'Customer', 'id',
        'name');
        $obs         = new TText('obs');

        // cria os campos detalhe
        $product_detail_unqid   = new THidden('product_detail_unqid');
        $product_detail_id       = new THidden('product_detail_id');
        $product_detail_product_id = new TDBUniqueSearch('product_detail_product_id',
        'samples', 'Product', 'id', 'description');
        $product_detail_price    = new TEntry('product_detail_price');
        $product_detail_amount   = new TEntry('product_detail_amount');
        $product_detail_discount = new TEntry('product_detail_discount');
        $product_detail_total    = new TEntry('product_detail_total');

        // ajusta propriedades
        $id->setEditable(false);
        $customer_id->setMinLength(1);
        $obs->setSize('100%', 80);
        $product_detail_product_id->setSize('100%');
        $product_detail_product_id->setMinLength(1);
        $product_detail_price->setSize('100%');
        $product_detail_amount->setSize('100%');
        $product_detail_discount->setSize('100%');

        // validações de campos
        $date->addValidation('Date', new TRequiredValidator);
        $customer_id->addValidation('Customer', new TRequiredValidator);

        // ação de saída do produto
        $product_detail_product_id->setChangeAction(new
        TAction([$this, 'onProductChange']));
    }
}
```

```

// coloca campos mestre no formulário
$this->form->addFields( [new TLabel('ID')], [$id],
    [new TLabel('Date (*)', '#FF0000')], 
    [$date] );

$this->form->addFields( [new TLabel('Customer (*)', '#FF0000')], 
    [$customer_id] );

$this->form->addFields( [new TLabel('Obs')], [$obs] );

// adiciona os campos de detalhe no formulário
$this->form->addContent( ['<h4>Details</h4><br>'] );
$this->form->addFields( [$product_detail_unqid], [$product_detail_id] );
$this->form->addFields( [ new TLabel('Product (*', '#FF0000') ],
    [$product_detail_product_id],
    [ new TLabel('Amount(*)', '#FF0000') ],
    [$product_detail_amount] );
$this->form->addFields( [ new TLabel('Price (*', '#FF0000') ],
    [$product_detail_price],
    [ new TLabel('Discount') ],
    [$product_detail_discount] );

// cria botão para adicionar produto
$add_product = TButton::create('add_product', [$this, 'onProductAdd'],
'Register', 'fa:plus-circle green');
$add_product->getAction()->setParameter('static','1');
$this->form->addFields( [], [$add_product] );

// cria adatagrid para o detail
$this->product_list = new BootstrapDatagridWrapper(new TDataGrid);
$this->product_list->setHeight(150);
$this->product_list->makeScrollable();
$this->product_list->setId('products_list');
$this->product_list->generateHiddenFields();
$this->product_list->style = "min-width: 700px; width:100%;margin-bottom: 10px";

// cria as colunas para detail
$col_uniq = new TDataGridColumn( 'unqid', 'Uniqid', 'center', '10%' );
$col_id = new TDataGridColumn( 'id', 'ID', 'center', '10%' );
$col_pid = new TDataGridColumn( 'product_id', 'ProdID', 'center', '10%' );
$col_descr = new TDataGridColumn( 'product_id', 'Product', 'left', '30%' );
$col_amount = new TDataGridColumn( 'amount', 'Amount', 'left', '10%' );
$col_price = new TDataGridColumn( 'sale_price', 'Price', 'right', '15%' );
$col_disc = new TDataGridColumn( 'discount', 'Discount', 'right', '15%' );
$col_subt = new TDataGridColumn( '{amount} * ({sale_price} - {discount})',
'Subtotal', 'right', '20%' );

// adiciona as colunas à datagrid de detail
$this->product_list->addColumn( $col_uniq );
$this->product_list->addColumn( $col_id );
$this->product_list->addColumn( $col_pid );
$this->product_list->addColumn( $col_descr );
$this->product_list->addColumn( $col_amount );
$this->product_list->addColumn( $col_price );
$this->product_list->addColumn( $col_disc );
$this->product_list->addColumn( $col_subt );

// transforma o código do produto em sua descrição
$col_descr->setTransformer(function($value) {
    return Product::findInTransaction('samples', $value)->description;
});
// desliga visibilidade destas colunas
$col_id->setVisibility(false);
$col_uniq->setVisibility(false);

```

```

// cria ação para editar o item, passando todos atributos (*)
$action1 = new TDataGridAction([$_this, 'onEditItemProduto']);
$action1->setFields( ['unqid', '*' ] );

// cria ação para excluir o item da tela, passando o seu unqid
$action2 = new TDataGridAction([$_this, 'onDeleteItem']);
$action2->setField('unqid');

// adiciona ações nadatagrid
$_this->product_list->addAction($action1, _t('Edit'), 'fa:edit blue');
$_this->product_list->addAction($action2, _t('Delete'), 'fa:trash red');

// cria estrutura em memória
$_this->product_list->createModel();

// cria painel ao redor dadatagrid de detail
$panel = new TPanelGroup;
$panel->add($_this->product_list);
$panel->getBody()->style = 'overflow-x:auto';
$_this->form->addContent( [$panel] );

// declara uma função de transformação
$format_value = function($value) {
    if (is_numeric($value)) {
        return 'R$ '.number_format($value, 2, ',', '.');
    }
    return $value;
};
// aplica transformers de valor monetário
$c1_col_price->setTransformer( $format_value );
$c1_col_disc->setTransformer( $format_value );
$c1_col_subt->setTransformer( $format_value );

// cria ação para fechar a janela
$_this->form->addActionLink( _t('Close'),
    new TAction([__CLASS__, 'onClose'], ['static'=>'1']), 'fa:times red');

// cria ação para salvar o registro, static=1 é para não recarregar a página
$_this->form->addAction( 'Save',
    new TAction([$_this, 'onSave'], ['static'=>'1']), 'fa:save green');

// cria ação para limpar o formulário
$_this->form->addAction( 'Clear',
    new TAction([$_this, 'onClear']), 'fa:eraser red');
parent::add($_this->form);
}

```

O formulário terá um método a ser executado sempre que o usuário selecionar um produto, que será o `onProductChange()`. Este método receberá um código de produto (`product_detail_product_id`), e deve buscar na base de dados o preço correspondente para preencher automaticamente o campo de preço sempre que o usuário selecionar um produto. Para tal, ele abre uma transação com a base de dados, carrega o produto correspondente (`new Produto`), envia o preço de venda para o formulário. Para enviar o preço de venda, é usado o método `TForm::sendData()`, que recebe o nome do formulário, e um objeto contendo os dados a serem enviados. Este objeto é declarado de maneira inline (`object`) a partir de um um vetor indexado pelo preço de venda (`product_detail_price`). Neste caso, estamos enviando somente o atributo `product_detail_price`, mas poderíamos enviar outros atributos também.

```

public static function onProductChange( $params )
{
    if( !empty($params['product_detail_product_id']) )
    {
        try {
            TTransaction::open('samples');
            $product = new Product($params['product_detail_product_id']);
            TForm::sendData('form_Sale', (object)
                ['product_detail_price' => $product->sale_price ]);
            TTransaction::close();
        }
        catch (Exception $e) {
            new TMessage('error', $e->getMessage());
            TTransaction::rollback();
        }
    }
}

```

Sempre que o usuário limpar o formulário, clicando no botão, o método `onClear()` será executado. Este método chama o método `clear()` do formulário, para limpá-lo.

```

function onClear($param)
{
    $this->form->clear();
}

```

Sempre que o usuário clicar para adicionar um produto nadatagrid de detalhes, o método `onProductAdd()` será executado. Este método obtém os dados do formulário, e verifica se o usuário preencheu alguns atributos obrigatórios. Caso algum campo tenha ficado vazio, este método emite uma exceção.

Em seguida, é gerado um ID único (`uniqid`) para a nova linha de dados. Caso seja uma edição, o `uniqid` já tem valor e neste caso é mantido. A variável `$grid_data` é alimentada com todos os dados a serem inseridos nadatagrid de detail. O método `addItem()` dadatagrid, recebe estes dados, e gera uma linha (`$row`), que é inserida nadatagrid pelo método `replaceRowById()`. Este método recebe um `uniqid`, e a linha inteira (`$row`). Caso seja uma edição, a linha é substituída (o `uniqid` já existe), caso contrário, ela é adicionada, pois o `$uniqid` será totalmente novo neste caso. Por fim, todos os campos do detail são limpos, e enviados para o formulário pelo `sendData()`.

```

public function onProductAdd( $param )
{
    try {
        $this->form->validate();
        $data = $this->form->getData();

        // realiza algumas validações
        if( (! $data->product_detail_product_id) ||
            (! $data->product_detail_amount) ||
            (! $data->product_detail_price) )
        {
            throw new Exception('The fields Product, Amount and Price required');
        }
        // usa o mesmo unqid, ou gera um novo se vazio
        $uniqid = !empty($data->product_detail_uniqid) ? $data-
>product_detail_uniqid : uniqid();
    }
}

```

```

// vetor com dados para novo detail
$grid_data = ['uniqid'      => $uniqid,
              'id'          => $data->product_detail_id,
              'product_id'   => $data->product_detail_product_id,
              'amount'       => $data->product_detail_amount,
              'sale_price'   => $data->product_detail_price,
              'discount'     => $data->product_detail_discount];

// gera uma linha nova
$row = $this->product_list->addItem( object $grid_data );
$row->id = $uniqid;

// substitui ou insere a nova linha (uniqid controla isto)
TDataGrid::replaceRowById('products_list', $uniqid, $row);

// limpa campos do detail
$data->product_detail_uniqid    = '';
$data->product_detail_id         = '';
$data->product_detail_product_id = '';
$data->product_detail_name       = '';
$data->product_detail_amount     = '';
$data->product_detail_price      = '';
$data->product_detail_discount   = '';

// envia dados para formulário
TForm::sendData( 'form_Sale', $data, false, false );
}
catch (Exception $e) {
    $this->form->setData( $this->form->getData());
    new TMessage('error', $e->getMessage());
}
}

```

Sempre que o usuário clicar em um item da datagrid para editá-lo, o método `onEditItemProduto()` é executado. Este método recebe pela variável `$param`, todos os dados do detail dadatagrid. Com base nestes dados, é montado um objeto (`$data`), com todos atributos necessários para preencher os campos do formulário de detail. Por fim, usamos o método `sendData()` para enviar estes dados para os respectivos campos.

```

public static function onEditItemProduto( $param )
{
    $data = new stdClass;
    $data->product_detail_uniqid    = $param['uniqid'];
    $data->product_detail_id         = $param['id'];
    $data->product_detail_product_id = $param['product_id'];
    $data->product_detail_amount     = $param['amount'];
    $data->product_detail_price      = $param['sale_price'];
    $data->product_detail_discount   = $param['discount'];
    // envia dados, sem disparar os métodos change/exit
    TForm::sendData( 'form_Sale', $data, false, false );
}

```

Sempre que o usuário clicar sobre um item dadatagrid de detalhes para exclui-lo, será necessário remover a respectiva linha dadatagrid, e limpar o formulário com os campos de detail. Inicialmente criamos um objeto (`$data`), contendo em seus atributos todos os nomes dos campos que queremos limpar. Então, usamos o `sendData()` para enviar os dados para os campos. Por fim, usamos o método `removeRowById()` para remover a linha dadatagrid. Este método recebe o id dadatagrid, e o `uniqid` da linha para remover.

```

public static function onDeleteItem( $param )
{
    $data = new stdClass;
    $data->product_detail_uniqid      = '';
    $data->product_detail_id          = '';
    $data->product_detail_product_id = '';
    $data->product_detail_amount     = '';
    $data->product_detail_price      = '';
    $data->product_detail_discount   = '';

    // send data, do not fire change/exit events
    TForm::sendData( 'form_Sale', $data, false, false );

    // remove row
    TDataGrid::removeRowById('products_list', $param['uniqid']);
}

```

Sempre que uma venda for editada, o que acontece a partir de uma datagrid (que será criada no próximo exemplo), o método `onEdit()` é executado. Este método deve carregar em tela tanto os dados da venda, quanto de seus itens. Para tal, este método lê o parâmetro `key`, que identifica a chave da venda a ser editada, e carrega para a memória a venda (`new Sale`), e os itens da venda, pelo método `SaleItem::where(...)->load()`. Os itens da venda são então percorridos, e cada item é adicionado nadatagrid pelo seu método `addItem()`. Antes, no entanto, é importante gerar um uniqid para aquela linha, a fim de distingui-la das demais em ações de edição e exclusão. O método `setData()` joga os dados do objeto lido do banco de dados para o formulário em tela.

```

function onEdit($param)
{
    try {
        TTransaction::open('samples');

        if (isset($param['key']))
        {
            $key = $param['key'];

            $object = new Sale($key);
            $sale_items = SaleItem::where('sale_id', '=', $object->id)->load();

            foreach( $sale_items as $item )
            {
                $item->uniqid = uniqid();
                $row = $this->product_list->addItem( $item );
                $row->id = $item->uniqid;
            }
            $this->form->setData($object);
            TTransaction::close();
        }
        else
        {
            $this->form->clear();
        }
    }
    catch (Exception $e) {
        new TMessage('error', $e->getMessage());
        TTransaction::rollback();
    }
}

```

Sempre que o usuário clicar no botão para salvar a venda, não somente o registro principal, mas também os itens da venda deverão ser salvos. Os dados da venda virão a partir do formulário (`$data`). Sempre que o botão para salvar for clicado, é criado um objeto `Sale`, e este é alimentado a partir dos dados submetidos (`$data`). Caso nestes dados tenha um ID, trata-se de uma edição, caso contrário, trata-se de um registro novo.

Sempre que uma venda for salva, seus itens são excluídos antes de serem reinseridos, o que é feito pelo método `SaleItem::where(...)->delete()`. Em seguida, os dados do detalhamento são percorridos. Os dados do detalhamento são armazenados em campos escondidos na própria datagrid, o que somente ocorre pois utilizamos o método `generateHiddenFields()`. Por padrão, o nome desses campos escondidos é formado pelo ID da datagrid (`products_list`) concatenado com o nome da coluna em si (Ex: `product_id`). Como estes campos são vetoriais [], eles podem ser percorridos por um `foreach`. Para cada item de venda, um objeto `SaleItem` é instanciado, alimentado a partir deste campo vetorial e armazenado na base por meio do método `store()`. Antes, no entanto, o total da venda é acumulado para se atualizado ao final.

```

function onSave()
{
    try {
        TTransaction::open('samples');

        $data = $this->form->getData();
        $this->form->validate();

        // cria a venda, e alimenta a partir dos dados do formulário
        $sale = new Sale();
        $sale->fromArray((array) $data);
        $sale->store();

        // apaga os itens
        SaleItem::where('sale_id', '=', $sale->id)->delete();

        // percorre os items a partir da tela
        if( $param['products_list_product_id'] ) {
            $total = 0;
            foreach( $param['products_list_product_id'] as $key => $item_id ) {
                $item = new SaleItem();
                $item->product_id = $item_id;
                $item->sale_price = (float) $param['products_list_sale_price'][$key];
                $item->amount = (float) $param['products_list_amount'][$key];
                $item->discount = (float) $param['products_list_discount'][$key];
                $item->total = ($item->sale_price * $item->amount) - $item->discount;

                $item->sale_id = $sale->id;
                $item->store();
                $total += $item->total;
            }
        }
        $sale->total = $total;
        $sale->store(); // atualiza a venda com o total
        // envia o ID da venda para a tela, caso o usuário salve novamente.
        TForm::sendData('form_Sale', (object) ['id' => $sale->id]);

        TTransaction::close(); // fecha transação
        new TMessage('info', 'Registro salvo');
    }
}

```

```

        catch (Exception $e)
    {
        new TMessage('error', $e->getMessage());
        $this->form->setData( $this->form->getData() ); // mantém o form preenchido
        TTransaction::rollback();
    }
}

// método para fechar janela
public static function onClose()
{
    parent::closeWindow();
}
}

```

5.3.6 Listagem de vendas

Para completar nosso exemplo sobre vendas, vamos criar umadatagrid para localizar vendas, filtrando por diferentes campos como código, cliente, e intervalo de datas da venda. Para desenvolver esta tela, utilizaremos somente os métodos padrão fornecidos pelo Trait `AdiantiStandardListTrait`, o que facilitará muito a construção desta tela. A figura a seguir demonstra o resultado final da listagem.

ID	Date	Customer	Total
1	01/07/2018	Andrei Zmievski	R\$ 505,00
2	02/07/2018	Rubens Prates	R\$ 1.945,00
3	03/07/2018	Augusto Campos	R\$ 4.880,00
4	04/07/2018	Marcelio Leal	R\$ 1.060,00
5	05/07/2018	Manuel Lemos	R\$ 1.890,00
6	06/07/2018	Fábio Locatelli	R\$ 12.900,00
7	07/07/2018	Leonardo Soldatelli	R\$ 620,00
8	08/07/2018	Alberto Bengoa	R\$ 495,00
9	09/07/2018	Andrei Zmievski	R\$ 79,00
10	10/07/2018	Marcelio Leal	R\$ 40,00

Figura 124 Listagem de vendas

A listagem de vendas começa com a importação do Trait `AdiantiStandardListTrait`, que nos fornecerá métodos muito úteis como `onDelete()`, `onReload()`, `addFilterField()`, e outros, o que permitirá construir a listagem com poucas linhas de código.

O método `setDatabase()` indica a base de dados, enquanto que o `setActiveRecord()` a classe a ser manipulada. O método `addFilterField()` adiciona um campo de filtro,

indicando o nome do campo do banco, o operador de comparação, e o nome do campo da tela. Este método aceita opcionalmente um quarto parâmetro que é uma função anônima para transformação do dado que está no formulário antes de ir para a consulta do banco (filtro). No caso das datas, que estão no formato brasileiro em tela, usamos a função de transformação para convertê-las no formato americano antes de serem utilizadas como filtro, por meio da função `convertToMask()`.

Em seguida, criamos o formulário (`BootstrapFormBuilder`), os campos do formulário, e adicionamos os campos ao formulário pelo método `addField()`. Em seguida, configuraremos algumas propriedades dos campos como tamanho, pelo método `setSize()`, e no caso das datas a máscara, pelo método `setMask()`.

O método `setData()` é usado para manter o formulário preenchido com os dados da busca, que por padrão são armazenados na variável de sessão `SaleList_filter_data`.

Em seguida, é criada a datagrid (`TDataGrid`), e suas colunas (`TGridColumn`). A função `$format_value` é criada para aplicar formatação monetária na coluna total, por meio do método `setTransformer()`.

As colunas de id e data, terão ainda a função de reordenação, o que é definido pelo método `setAction()`, chamando a função `onReload()` com outro parâmetro `order`.

A coluna de data também terá uma função de transformação, que fará com que a coluna seja exibida no formato brasileiro, pelo método `setTransformer()`.

A datagrid terá duas ações: editar, que estará vinculada ao método `onEdit()` da classe `SaleForm` (desenvolvida anteriormente); e excluir, que estará vinculada ao método `onDelete()` do Trait importado.

app/control/Organization/ComplexViews/SaleList.class.php

```
<?php
class SaleList extends TPage
{
    protected $form, $datagrid, $pageNavigation;

    // importa Trait com métodos onReload, onDelete, etc.
    use Adianti\Base\AdiantiStandardListTrait;

    public function __construct()
    {
        parent::__construct();
        $this->setDatabase('samples');           // define o banco
        $this->setActiveRecord('Sale');          // define a classe
        $this->setDefaultOrder('id', 'asc');     // define a ordem

        // configura campos de filtro do formulário em relação ao banco
        $this->addFilterField('id', '=', 'id'); // campo, operador, campo form
        $this->addFilterField('customer_id', '=', 'customer_id');

        $this->addFilterField('date', '>=', 'date_from', function($value) {
            return TDate::convertToMask($value, 'dd/mm/yyyy', 'yyyy-mm-dd');
        }); // campo, operador, campo do form, transformação
```

```

$this->addFilterField('date', '<=', 'date_to', function($value) {
    return TDate::convertToMask($value, 'dd/mm/yyyy', 'yyyy-mm-dd');
}); // campo, operador, campo do form, transformação

// cria o formulário
$this->form = new BootstrapFormBuilder('form_search_Sale');
$this->form->setFormTitle(_t('Sale list'));

// cria os campos do formulário
$id = new TEntry('id');
$date_from = new TDate('date_from');
$date_to = new TDate('date_to');
$customer_id = new TDBUniqueSearch('customer_id', 'samples', 'Customer', 'id',
'name');
$customer_id->setMinLength(1);
$customer_id->setMask('{name} {id}');

// adiciona campos ao formulário
$this->form->addField([new TLabel('Id')], [$id]);
$this->form->addField([new TLabel('Date (from)'), $date_from],
[new TLabel('Date (to)'), $date_to]);
$this->form->addField([new TLabel('Customer')], [$customer_id]);

// define propriedades dos campos
$id->setSize('50%');
$date_from->setSize('100%');
$date_to->setSize('100%');
$date_from->setMask('dd/mm/yyyy');
$date_to->setMask('dd/mm/yyyy');

// mantém formulário preenchido com dados da busca
$this->form->setData(TSession::getValue('SaleList_filter_data'));

// adiciona ações ao formulário
$this->form->addAction('Find', new TAction([$this, 'onSearch']), 'fa:search');
$this->form->addActionLink('New', new TAction(['SaleForm', 'onEdit']),
'bs:plus-sign green');

// cria adatagrid
$this->datagrid = new BootstrapDatagridWrapper(new TDataGrid);
$this->datagrid->width = '100%';

// cria as colunas dadatagrid
$column_id = new TDataGridColumn('id', 'Id', 'center', '10%');
$column_date = new TDataGridColumn('date', 'Date', 'center', '20%');
$column_customer = new TDataGridColumn('customer->name', 'Customer', 'left',
'50%');
$column_total = new TDataGridColumn('total', 'Total', 'right', '20%');

// cria função de formatação
$format_value = function($value) {
    if (!is_numeric($value)) {
        return 'R$ ' . number_format($value, 2, ',', '.');
    }
    return $value;
};

$column_total->setTransformer($format_value);

// adiciona as colunas nadatagrid
$this->datagrid->addColumn($column_id);
$this->datagrid->addColumn($column_date);
$this->datagrid->addColumn($column_customer);
$this->datagrid->addColumn($column_total);

```

```

// atribui funções de ordenação nas colunas
$column_id->setAction(new TAction([$this, 'onReload']), ['order' => 'id']);
$column_date->setAction(new TAction([$this, 'onReload']), ['order' => 'date']);

// aplica função de transformação de data
$column_date->setTransformer( function($value, $object, $row) {
    $date = new DateTime($value);
    return $date->format('d/m/Y');
});

// cria ação para edição de registro
$action_edit = new TDataGridAction(['SaleForm', 'onEdit'], ['key' => '{id}']);
// cria ação para exclusão de registro
$action_delete = new TDataGridAction([$this, 'onDelete'], ['key' => '{id}']);
// adiciona ações
$this->datagrid->addAction($action_edit, 'Edit', 'fa:edit blue fa-fw');
$this->datagrid->addAction($action_delete, 'Delete', 'fa:trash red fa-fw');

// cria a estrutura dadatagrid
$this->datagrid->createModel();

// cria o paginador
$this->pageNavigation = new TPageNavigation;
$this->pageNavigation->setAction(new TAction([$this, 'onReload']));
$this->pageNavigation->setWidth($this->datagrid->getWidth());

// cria um container vertical
$container = new TVBox;
$container->style = 'width: 100%';
$container->add($this->form);
$container->add(TPanelGroup::pack('', $this->datagrid, $this->pageNavigation));

parent::add($container);
}
}

```

5.4 Telas de consulta

5.4.1 Consulta o status de um cliente

No capítulo 4, vimos como criar um Template view, que nos permite mesclar HTML com dados dinâmicos para compor uma interface. Nesse exemplo, vamos usar um Template para exibir dados de um cliente, bem como suas compras efetuadas. Podemos usar um Template sempre que precisarmos exibir um conjunto heterogêneo de informações em interface, quando os componentes padrão do Framework não forem suficientes para apresentar este conjunto de informações, ou quando quisermos oferecer ao usuário uma interface mais elaborada, com diferentes estilos de formatação.

Neste exemplo, criaremos um formulário para busca de clientes. Sempre que o usuário localizar um cliente, poderá clicar no botão "Check status", que apresentará, por meio de um template, os dados do cliente, bem como as suas compras efetuadas.

No código a seguir, temos alguns trechos do HTML a serem utilizados como modelo para apresentar os dados do cliente em tela. Temos a seção `main`, `sale-details`, que contém os dados da venda, e `sale-totals`, contendo os totais.

```
app/resources/customer_status.html
<!--[main]-->
<table>
    <tr>
        <td>
            <table class="customform" style="width:640px">
                <tr>
                    <td class="sectiontitle" colspan=4>Customer data</td>
                </tr>
                <tr bgcolor="#e0e0e0">
                    <td style="font-weight:bold" align="center">ID</td>
                    <td style="font-weight:bold" align="center" colspan="2">Name</td>
                </tr>
                <tr>
                    <td align="center">{$id}</td>
                    <td align="center" colspan="2"><a href="#">{$name}</a></td>
                </tr>
            </table>
        </td>
    </tr>
    <tr>
        <td>
            <table class="customform" style="width:640px">
                <tr>
                    <td class="sectiontitle" colspan="3">Sales</td>
                </tr>
                <tr bgcolor="#e0e0e0">
                    <td align="center">Date</td>
                    <td align="center">Product</td>
                    <td align="center">Sale Price</td>
                </tr>
                <!--[sale-details]-->
                <tr>
                    <td align="center">{$date}</td>
                    <td align="left">{$product_id} - {$product_description}</td>
                    <td align="right">{$sale_price}</td>
                </tr>
                <!--[/sale-details]-->
                <!--[sale-totals]-->
                <tr bgcolor="#CECECE">
                    <td align="right" colspan="2"><b> Total: </b></td>
                    <td align="right"><b> {$total} </b></td>
                </tr>
                <!--[/sale-totals]-->
            </table>
        </td>
    </tr>
</table>
<!--[/main]-->
```

No método construtor, criamos o formulário (`BootstrapFormBuilder`), e adicionamos o campo de busca de clientes (`TDBUniqueSearch`) ao formulário. Também criamos um botão de ação, por meio do método `addAction()`. O botão de ação executará o método `onCheckStatus()`, para apresentar os resultados ao usuário. Teremos outro botão de ação que exportará a tela de consulta de status no formato PDF, vinculado ao método `onExportPDF()`, que utilizará a biblioteca DomPDF para a conversão.

app/control/Organization/ComplexViews/CustomerStatusView.class.php

```
<?php
class CustomerStatusView extends TPage
{
    private $form;

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

        // cria o formulário
        $this->form = new BootstrapFormBuilder('form');
        $this->form->setFormTitle(_t('Customer Status'));

        // cria o campo de busca de cliente
        $customer_id = new TDBUniqueSearch('customer_id', 'samples', 'Customer', 'id',
            'name', 'id');
        $customer_id->setMask('{name} ({id})');
        $customer_id->setMinLength(1);

        // adiciona o campo ao formulário
        $this->form->addFields( [new TLabel('Customer')], [$customer_id]);

        // adiciona ação para consultar o status
        $this->form->addAction('Check status', new TAction(array($this,
            'onCheckStatus')), 'fa:check-circle-o green');

        // ação para exportar tela em PDF
        $this->form->addAction('Export to PDF', new TAction(array($this,
            'onExportPDF')), 'fa:file-pdf-o red');

        parent::add($this->form);
    }
}
```

Sempre que o usuário clicar no botão de ação, o método `onCheckStatus()` será executado. Este método carrega o template, por meio da classe `THtmlRenderer`. Após, carrega o cliente indicado no formulário `new Customer`. Caso encontrado, o cliente será convertido em array pelo método `toArray()`, e complementado com outras informações, como `city_name`. Então, a seção `main` é habilitada pelo método `enableSection()`, passando o cliente (`$array_object`) como parâmetro para substituições.

Após habilitar a seção `main`, o método `getSales()` é utilizado para obter as vendas realizadas para aquele cliente. Para cada venda (`foreach $sales`), ainda executamos o método `getSaleItems()`, para obter os itens da venda. Para cada item de uma venda, acrescentamos suas informações em uma matriz (`$replaces`). Esta matriz será utilizada para preencher os detalhes (seção `sale-details` do HTML). É importante lembrar que a seção `sale-details` será repetida conforme a quantidade de itens que existirem na matriz `$replaces`.

Para cada iteração do `foreach`, somamos o total de itens. Por fim, habilitamos a seção `sale-details`, preenchendo-a com a variável `$replaces`. Aqui, é importante lembrar de passar o parâmetro `TRUE` ao final, para que o conteúdo da seção seja repetido, conforme a quantidade de itens da matriz.

```

public function onCheckStatus($param)
{
    try {
        $data = (object) $param;
        $this->form->setData($data); // mantém o formulário preenchido

        // carrega o Template
        $html = new THtmlRenderer('app/resources/customer_status.html');

        TTransaction::open('samples');

        if (isset($data->customer_id))
        {
            // carrega o cliente
            $object = new Customer($data->customer_id);

            if ($object)
            {
                // cria um vetor com os dados do cliente
                $array_object = $object->toArray();
                $array_object['city_name'] = $object->city_name;
                $array_object['category_name'] = $object->category_name;

                // habilita a seção main, passando o vetor de substituições
                $html->enableSection('main', $array_object);

                $replaces = array();
                $sales = $object->getSales();
                if ($sales)
                {
                    $total = 0;
                    // percorre as vendas do cliente
                    foreach ($sales as $sale)
                    {
                        // percorre os itens da venda
                        foreach ($sale->getSaleItems() as $item)
                        {
                            // define uma matriz com os itens da venda
                            $replaces[] = array(
                                'date' => $sale->date,
                                'product_id' => $item->product_id,
                                'product_description' => $item
                                    ->product->description,
                                'sale_price' => number_format($item->sale_price, 2) ,
                                'amount' => $item->amount,
                                'discount' => $item->discount,
                                'total' => number_format($item->total, 2)
                            );
                            $total += $item->total;
                        }
                    }
                    $totals['total'] = number_format($total, 2);

                    // substitui os itens da venda e os totais
                    $html->enableSection('sale-details', $replaces, true);
                    $html->enableSection('sale-totals', $totals);
                }
                // ...
            }
            else {
                throw new Exception('Customer not found');
            }
        }
    }
}

```

```
TTransaction::close();  
  
// adiciona o Template à página  
parent::add($html);  
  
// retorna o renderer  
return $html;  
}  
}  
catch(Exception $e) {  
    new TMessage('error', $e->getMessage());  
}  
}
```

O formulário de busca de clientes terá ainda um botão que permitirá exportar o Template em PDF, já contendo os dados do cliente, e das vendas realizadas. O método `onExportPDF()` inicia com a execução do método `onCheckStatus()`, que processa todo o Template, e retorna o objeto `THtmlRenderer`. Em seguida, é utilizado o método `getContents()` para obter o HTML pronto já processado.

Para realizar a conversão para PDF, utilizamos a classe DomPDF. Ela possui o método `loadHtml()`, que carrega o HTML, e o método `render()` que renderiza o PDF em memória. O método `output()` gera o PDF bruto, que é salvo em disco pela função `file_put_contents()`. Em seguida, é aberta uma janela para exibir o PDF ao usuário.

```

public function onExportPDF($param)
{
    try {
        // processa o Template e retorna o renderer
        $html = $this->onCheckStatus($param);

        // obtém o HTML como string
        $contents = $html->getContents();

        // converte o HTML em PDF
        $dompdf = new \Dompdf\Dompdf();
        $dompdf->loadHtml($contents);
        $dompdf->setPaper('A4', 'portrait');
        $dompdf->render();

        $file = 'app/output/status.pdf';

        // escreve em disco
        file_put_contents($file, $dompdf->output());

        // exibe em janela
        $window = TWindow::create(_t('Customer Status'), 0.8, 0.8);
        $object = new TElement('object');
        $object->data = $file;
        $object->type = 'application/pdf';
        $object->style = "width: 100%; height:calc(100% - 10px)";
        $window->add($object);
        $window->show();
    }
    catch (Exception $e) {
        new TMessage('error', $e->getMessage());
    }
}

```

5.5 Operações em lote

5.5.1 Edição de registros em lote

Neste exemplo, vamos criar uma datagrid em que uma das colunas será exibida na forma de um campo de entrada de dados (`TEntry`), permitindo a alteração de valores diretamente nadatagrid. Na figura a seguir, podemos conferir o resultado do programa que criaremos. É possível perceber o formulário de buscas no topo, e adatagridabaixo, com o campo para preço de venda editável.

Neste programa, assim que o usuário sair com o foco do cursor do campo de preço (Sale Price), o dado será instantaneamente atualizado na tabela de produtos. O usuário poderá utilizar a tecla TAB para sair do campo, sendo que neste caso, além do valor ser gravado no banco de dados, o cursor será posicionado no próximo campo.

O formulário de buscas acima dadatagrid se comportará como em outras datagrids, filtrando os dados a serem exibidos. A paginação abaixo dadatagrid também funcionará da mesma maneira e não afetará a edição de valores, pois a gravação ocorrerá imediatamente na saída do campo.

Lista de edição instantânea				
MARCOS ANTONIO RAFAEL DA FONSECA ~				
Description	<input type="text"/>			
<input type="button" value="Q Buscar"/>				
Id	Description	Unity	Stock	Sale Price
1	Pendrive 512Mb	PC	10.0	<input type="text" value="57.6"/>
2	HD 120 GB	PC	20.0	<input type="text" value="180.0"/>
3	SD CARD 512MB	PC	4.0	<input type="text" value="35.0"/>
4	SD CARD 1GB MINI	PC	3.0	<input type="text" value="40.0"/>
5	CAM. PHOTO I70 Silver	PC	5.0	<input type="text" value="900.0"/>
6	CAM. PHOTO DSC-W50 Silver	PC	4.0	<input type="text" value="700.0"/>
7	WEBCAM INSTANT VF0040SP	PC	4.0	<input type="text" value="80.0"/>
8	CPU 775 CEL.D 360 3.46 533M	PC	10.0	<input type="text" value="300.0"/>
9	Recorder DCR-DVD108	PC	2.0	<input type="text" value="1,400.0"/>
10	HD IDE 80G 7.200	PC	8.0	<input type="text" value="160.0"/>

MARCOS ANTONIO RAFAEL DA FONSECA ~

Figura 125 Edição de registros em lote

O programa inicia com a importação do Trait `AdiantiStandardListTrait`, que fornecerá os métodos básicos para filtro, ordenação e paginação. O método construtor inicia com a definição de base de dados, por meio do método `setDatabase()`, a definição da classe manipulada, por meio do método `setActiveRecord()`, e a ordenação default, por meio do método `setDefaultOrder()`. O método `addFilterField()` adiciona uma definição de filtro, relacionando campo do banco com campo do formulário.

Após, o formulário (`BootstrapFormBuilder`) é instanciado. Em seguida, é instanciado o campo de busca (`TEntry`), e é adicionado ao formulário. O método `setData()` mantém o formulário preenchido com os dados da última busca.

Em seguida, é criada a datagrid (`TDataGrid`), e são criadas as colunas (id, descrição, unidade, estoque, preço de venda). A coluna de preço (`$column_sale_price`) será transformada para em seu lugar ser exibido um input `TEntry` para digitação do valor.

A transformação da coluna de preço de venda ocorre pelo método `setTransformer()`, que recebe como parâmetros o valor da coluna onde ele foi aplicado (`$value`), todo objeto de dados adicionado naquela linha (`$object`), e a linha criada no HTML (`$row`). Para realizar a transformação, é instanciado um objeto `TEntry`, cujo nome terá como prefixo `sale_price`, seguido do `id` do objeto de dados. Este objeto terá como valor default o preço de venda, o que é definido pelo método `setValue()`. O método `setNumericMask()` liga a máscara de digitação numérica. Este objeto terá uma ação de saída de campo, o que é definido pelo método `setExitAction()`. Sempre que o usuário sair do campo, o método `onSaveInline()` da própria classe será executado. Este método receberá, além do próprio input, o nome da coluna e a descrição do produto editado. O objetivo deste método será gravar o preço na base de dados.

Após a transformação do campo, as colunas são adicionadas à datagrid por meio do método `addColumn()`, e a sua estrutura é criada em memória, por meio do método `createModel()`. Em seguida é criado o objeto de paginação, e a interface final é montada por meio de uma caixa vertical (`TVBox`).

app/control/Organization/BatchOperations/ProductInstantUpdateList.class.php

```
<?php
class ProductInstantUpdateList extends TPage
{
    protected $datagrid;
    protected $pageNavigation;

    use Adianti\Base\AdiantiStandardListTrait;

    public function __construct()
    {
        parent::__construct();
        $this->setDatabase('samples');
        $this->setActiveRecord('Product');
        $this->setDefaultOrder('id', 'asc');
        // define o filtro (campo do banco, operador, campo do form)
        $this->addFilterField('description', 'like', 'description');
```

```

// cria o formulário
$this->form = new BootstrapFormBuilder('form_search_Product');
$this->form->setFormTitle(_t('Batch instant update list'));

// cria o campo de buscas
getDescription = new TEntry('description');
$this->form->addField( [new TLabel('Description')], [$description] );

// cria o botão de buscas
$this->form->addAction(_t('Find'), new TAction(array($this, 'onSearch')),
'fa:search');

// mantém o formulário preenchido com os dados da última busca
$this->form->setData( TSession::getValue('Product_filter_data') );

// cria adatagrid
$this->datagrid = new BootstrapDatagridWrapper(new TDataGrid);
$this->datagrid->style = 'width: 100%';
$this->datagrid->disableDefaultClick();

// cria as colunas dadatagrid
$column_id = new TDataGridColumn('id', 'Id', 'left');
$column_description = new TDataGridColumn('description', 'Description', 'left');
$column_unity = new TDataGridColumn('unity', 'Unity', 'center');
$column_stock = new TDataGridColumn('stock', 'Stock', 'right');
$column_sale_price = new TDataGridColumn('sale_price_widget', 'Sale Price',
'right');

// cria um campo para edição de preço, com uma ação de saída (exit)
$column_sale_price->setTransformer( function($value, $object, $row) {
    $widget = new TEntry('sale_price' . '_' . $object->id);
    $widget->setValue( $object->sale_price );
    $widget->setNumericMask(1, '.', ',');
    $widget->setSize(120);
    $widget->setFormName('form_search_Product');

    // define a ação de saída do campo
    $action = new TAction( [$this, 'onSaveInline'],
        ['column' => 'sale_price',
        'product_description' => $object->description] );

    $widget->setExitAction( $action );
    return $widget;
});
$this->datagrid->addColumn($column_id);
$this->datagrid->addColumn($column_description);
$this->datagrid->addColumn($column_unity);
$this->datagrid->addColumn($column_stock);
$this->datagrid->addColumn($column_sale_price);

// cria estrutura dadatagrid
$this->datagrid->createModel();

// cria o paginador
$pageNavigation = new TPageNavigation();
$pageNavigation->setAction(new TAction(array($this, 'onReload')));
$pageNavigation->setWidth($this->datagrid->getWidth());

$container = new TVBox;
$container->style = 'width: 100%';
$container->add(new XMLBreadCrumb('menu.xml', __CLASS__));
$container->add($this->form);
$container->add(TPanelGroup::pack('', $this->datagrid, $this->pageNavigation));

parent::add($container);
}

```

A ação de saída do campo de preço de venda estará conectada ao método `onSaveInline()`. O objetivo deste método é atualizar o preço de venda do produto na base de dados. Os métodos deste tipo (exit action) sempre recebem como parâmetro o nome do campo gerador do evento (`_field_name`), o valor do campo gerador do evento (`_field_value`), o nome do formulário gerador (`_form_name`), além de outros parâmetros programados pelo usuário como é o caso do parâmetro `column`.

O método `onSaveInline()` recebe estes dados, e a partir do nome do campo extrai o último segmento por meio de um `explode()` para obter o `$id` do campo editado. A partir do seu `id`, o objeto é carregado da base de dados, por meio do método `find()`. Em seguida o valor da coluna (`$column`) é atualizado no registro, o objeto é salvo por meio do `store()`, e uma mensagem é exibida ao usuário por meio do componente `TToast`.

```
public static function onSaveInline($param)
{
    $name = $param['_field_name'];
    $value = $param['_field_value'];
    $column = $param['column'];

    $parts = explode('_', $name);
    $id = end($parts);

    try
    {
        // abre transação
        TTransaction::open('samples');

        $object = Product::find($id);
        if ($object)
        {
            // atualiza atributo
            $object->$column = $value;

            // grava objeto
            $object->store();
        }

        // exibe mensagem de sucesso
        TToast::show('success', '<b>' . $param['product_description'] .
                    '</b> updated',
                    'bottom center', 'fa:check-circle-o');

        // fecha transação
        TTransaction::close();
    }
    catch (Exception $e)
    {
        // exibe mensagem de exceção
        TToast::show('error', $e->getMessage(), 'bottom center',
                    'fa:exclamation-triangle');
    }
}
```

5.5.2 Exclusão de registros em lote

Neste exemplo, vamos criar umadatagrid cujo objetivo é permitir a exclusão de registros em lote, permitindo ao usuário selecionar vários registros antes de clicar no botão para excluir os registros selecionados. Os registros selecionados são armazenados temporariamente em uma variável de sessão, para ao final serem lidos e excluídos. Nestadatagrid também implementaremos algumas facilidades como o clique em qualquer lugar da linha para marcar/desmarcar o registro atual.

Nadatagrid criada, teremos ainda a possibilidade de realizarmos uma busca, trazendo somente os registros filtrados, bem como a paginação na parte inferior. Na figura a seguir, podemos conferir o resultado do programa que criaremos. É possível perceber o formulário de buscas no topo, e adatagrid abaixo. Adatagrid terá uma ação apenas, que implementará a seleção de registros no clique da linha. O ícone desta ação será exibido como checkbox marcado ou vazio, conforme o clique do usuário. Um clique marca o registro, e o seguinte desmarca.

Como utilizaremos uma variável de sessão para armazenar os registros selecionados, poderemos selecionar registros de páginas diferentes. A paginação não afetará a seleção.

ID	Content
<input checked="" type="checkbox"/> 1	Content #1
<input type="checkbox"/> 2	Content #2
<input checked="" type="checkbox"/> 3	Content #3
<input type="checkbox"/> 4	Content #4
<input type="checkbox"/> 5	Content #5
<input type="checkbox"/> 6	Content #6
<input type="checkbox"/> 7	Content #7
<input type="checkbox"/> 8	Content #8
<input type="checkbox"/> 9	Content #9
<input type="checkbox"/> 10	Content #10

Figura 126 Exclusão de registros em lote

Esta tela de exclusão em lote utilizará o Trait `AdiantiStandardListTrait`. Dessa maneira, já incorporará uma série de funcionalidades para manipulação de listagens, tais como carregamento, ordenação, paginação e filtro. No método construtor, informamos qual a base de dados será utilizada, por meio do método `setDatabase()`, a classe a ser manipulada, por meio do método `setActiveRecord()`, e qual campo de busca será utilizado, por meio do método `addFilterField()`. Este exemplo terá não somente uma datagrid, mas também um formulário de filtro acima dela, permitindo localizar registros antes da exclusão. O exemplo utilizará a classe `TrashItem`, criada para esta demonstração. O método `setLimit()` define quantos registros serão exibidos por página.

O exemplo prossegue com a criação do formulário de buscas (`BootstrapFormBuilder`). Após a criação, é acrescentado um campo para buscas por conteúdo (`content`), que permitirá localizar os registros. O formulário de buscas terá uma ação vinculada ao método `onSearch()`, que é definido no `addAction()`. Este método vem do Trait.

Logo após o formulário de buscas, criamos adatagrid (`TDataGrid`). Em seguida, criamos as colunas dadatagrid (`TGridColumn`), e adicionamos estas àdatagrid pelo método `addColumn()`. Sobre a coluna id, será aplicado um transformer (`formatRow`). Este transformer será executado em cada uma das linhas dadatagrid e será responsável por preencher a cor de fundo da linha selecionada, e também alterar o ícone da ação que consta no início da linha (checkbox preenchida ou não).

Adatagrid terá uma ação de selecionar registro, vinculada ao método `onSelect()`, passando o id do registro como parâmetro. Após a definição da ação, é criada a estrutura dadatagrid em memória, por meio do método `createModel()`, e também é criada a paginação (`TPageNavigation`). Por fim, é criado um painel ao redor dadatagrid (`TPanelGroup`). Neste painel, será adicionada uma ação no cabeçalho por meio do método `addHeaderActionLink()`. Esta ação estará vinculada ao método `deleteSelected()` e seu objetivo é excluir os registros selecionados.

app/control/Organization/ComplexViews/MassiveDeleteView.class.php

```
<?php
class MassiveDeleteView extends TPage
{
    protected $form;
    protected $datagrid;
    protected $pageNavigation;

    // importa o trait com operações comuns para listagens
    use Adianti\Base\AdiantiStandardListTrait;

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

        $this->setDatabase('samples');
        $this->setActiveRecord('TrashItem');
        $this->addFilterField('content');
        $this->setLimit(10);
```

```

// cria o formulário
$this->form = new BootstrapFormBuilder('form_trash');
$this->form->setFormTitle(_t('Batch delete list'));

// cria os campos do formulário
$content = new TEntry('content');
$this->form->addField([new TLabel('Content')], [$content] );

// cria ação de busca de registros
$this->form->addAction('Search', new TAction([$this, 'onSearch']),
'fa:search');

// cria adatagrid
$this->datagrid = new BootstrapDatagridWrapper(new TDataGrid);
$this->datagrid->width = '100%';

// cria as colunas da datagrid
$Id      = new TDataGridColumn('id',      'ID',      'center', '10%');
$content = new TDataGridColumn('content', 'Content', 'left',   '90%');

// adiciona as colunas na datagrid
$this->datagrid->addColumn($Id);
$this->datagrid->addColumn($content);

// liga transformer na coluna id
$Id->setTransformer([$this, 'formatRow'] );

// cria ação para marcar/desmarcar o registro
$action1 = new TDataGridAction([$this, 'onSelect'],
['id' => '{id}', 'register_state' => 'false']);

$action1->setButtonClass('btn btn-default');

// adiciona ações na datagrid
$this->datagrid->addAction($action1, 'Select', 'fa:square-o fa-fw black');

// cria estrutura da datagrid em memória
$this->datagrid->createModel();

// cria o paginador
$pageNavigation = new TPageNavigation;
$this->pageNavigation->setAction(new TAction([$this, 'onReload']));

$panel = new TPanelGroup('');
$panel->add($this->datagrid);
$panel->addFooter($this->pageNavigation);

// adiciona ação para exclusão dos registros em lote
$panel->addHeaderActionLink('Delete selected',
new TAction([$this, 'deleteSelected']), 'fa:trash red' );

// caixa vertical
$container = new TVBox;
$container->style = 'width: 100%';
$container->add(new TXMLBreadCrumb('menu.xml', __CLASS__));
$container->add($this->form);
$container->add($panel);

parent::add($container);
}

```

Adatagrid terá uma ação para seleção de registros, vinculada ao método `onSelect()`. O objetivo desta ação é marcar/desmarcar registros, com base no clique da linha dadatagrid. Para saber quais registros estão ou não marcados, esta ação manterá uma variável de sessão (`__CLASS__.'selected_objects'`) contendo um vetor com todos os id's selecionados. Esta variável de sessão é lida por meio do método `TSession::getValue()`. Em seguida, é verificado se aquele id já existe neste vetor de sessão por meio de um `isset()`. Caso exista, o vetor é limpo naquela posição por meio de um `unset()`, caso contrário, é alimentado com o id. Em seguida, a variável de sessão é gravada novamente pelo método `TSession::setValue()`, e adatagrid recarregada.

```
public function onSelect($param)
{
    // lê o vetor da sessão, contendo os id's selecionados
    $selected_objects = TSession::getValue(__CLASS__.'selected_objects');

    $id = $param['id'];

    // verifica se o id já está no vetor
    if (isset($selected_objects[$id]))
    {
        unset($selected_objects[$id]);
    }
    else
    {
        $selected_objects[$id] = $id;
    }

    // atualiza a variável de sessão
    TSession::setValue(__CLASS__.'selected_objects', $selected_objects);

    // recarrega adatagrid
    $this->onReload( func_get_arg(0) );
}
```

Não basta que a ação `onSelect()` marque/desmarque os objetos em sessão. É necessário que alguma função crie um destaque visual para que o usuário saiba quais registros estão marcados. Este é o papel do método `formatRow()`, definido como transformer da coluna id. O objetivo de um transformer é modificar uma determinada coluna. Um transformer é aplicado sobre todas as linhas dadatagrid. Por padrão, o transformer recebe como parâmetros: o valor atual da coluna (`$value`), o objeto de dados completo (`$object`), e a linha criada para o HTML (`$row`), podendo utilizar quaisquer uma destas variáveis. Neste caso, utilizaremos a variável `$value`, já que esta conterá o id atual do registro dadatagrid, e utilizaremos também a variável `$row`, que representa a linha criada no HTML, para alterar sua cor de fundo por meio da definição de estilo (`style`).

Este método lê a variável de sessão com os id's selecionados, e verifica se o id atual (`$value`) está nesta lista. Caso esteja, a linha corrente tem seu fundo preenchido com uma cor (`$row->style`). Além disso, é utilizado o método `find()` para localizar o primeiro ícone (i) da linha, que na verdade é o ícone da ação. Ao encontrá-lo, o ícone é alterado para o ícone de checkbox preenchido (`check-square-o`), para dar o destaque necessário representando que aquele registro está marcado.

```

public function formatRow($value, $object, $row)
{
    $selected_objects = TSession::getValue(__CLASS__.'_'.$_selected_objects');

    if ($selected_objects)
    {
        if (in_array( (int) $value, array_keys( $selected_objects ) ) )
        {
            // altera a cor de fundo da linha
            $row->style = "background: #abdef9";

            // procura o primeiro ícone da linha
            $button = $row->find('i', ['class'=>'fa fa-square-o fa-fw black'])[0];

            if ($button)
            {
                // altera o ícone encontrado da ação
                $button->class = 'fa fa-check-square-o fa-fw black';
            }
        }
    }

    return $value;
}

```

Por fim, temos a ação que irá excluir os registros selecionados. O método `deleteSelected()` está vinculado a um botão posicionado na parte superior direita do painel (*Delete selected*). Este método lê a variável de sessão que contém os registros selecionados (`__CLASS__.'_'.$_selected_objects'`), percorre esses objetos, encontra-o um a um, por meio do método `find()`, e executa o método `delete()` sobre eles. Ao fim, a variável de sessão é reinicializada, e adatagrid recarregada por meio do método `onReload()`.

```

public function deleteSelected()
{
    $selected_objects = TSession::getValue(__CLASS__.'_'.$_selected_objects');

    if ($selected_objects)
    {
        TTransaction::open('samples');
        foreach ($selected_objects as $id)
        {
            $object = TrashItem::find($id);
            if ($object)
            {
                $object->delete();
            }
        }
        TTransaction::close();

        new TMessage('info', 'Records deleted');
    }
    TSession::setValue(__CLASS__.'_'.$_selected_objects', []);
    $this->onReload();
}

```

Obs: `TrashItem` é o nome da classe utilizada neste exemplo para representar os registros a serem excluídos. Ali, você deve alterar para o nome real (Produto, Cliente, Fornecedor, etc).

5.5.3 Seleção de registros em lote

É relativamente comum no desenvolvimento de aplicações de negócio precisarmos desenvolver interfaces que permitam ao usuário selecionar um conjunto de registros para que, sobre essa seleção, uma determinada ação seja executada. O próximo exemplo que desenvolveremos é justamente sobre isso: a construção de umadatagrid que permita seleção de múltiplos registros. Ao final, iremos somente exibir os registros selecionados. Porém, a partir do exemplo construído, você conseguirá facilmente alterá-lo para executar uma ação diferente.

Na figura a seguir, podemos conferir o resultado do programa que criaremos a seguir. É possível perceber o formulário de buscas no topo, e adatagridabaixo, com um botão para seleção de registro em cada linha. O botão “Show results” será responsável por abrir uma janela, exibindo os resultados selecionados.

Como este exemplo utilizará variáveis de sessão para armazenar os registros selecionados, o usuário poderá efetuar buscas ou paginar os resultados sem medo de perder a seleção atual de registros.

Lista de seleção em lote				
MARCOS ANTONIO RAFAEL DA FONSECA -				
Description				
<input type="text"/> Find				
Show results				
Id Description Sale Price				
<input checked="" type="checkbox"/>	1 Pendrive 512Mb	57.0		
<input type="checkbox"/>	2 HD 120 GB	180.0		
<input checked="" type="checkbox"/>	3 SD CARD 512MB	35.0		
<input type="checkbox"/>	4 SD CARD 1GB MINI	40.0		
<input type="checkbox"/>	5 CAM. PHOTO I70 Silver	900.0		
<input type="checkbox"/>	6 CAM. PHOTO DSC-W50 Silver	700.0		
<input type="checkbox"/>	7 WEBCAM INSTANT VF0040SP	80.0		
<input type="checkbox"/>	8 CPU 775 CEL.D 360 3.46 533M	300.0		
<input type="checkbox"/>	9 Recorder DCR-DVD108	1400.0		
<input type="checkbox"/>	10 HD IDE 80G 7.200	160.0		
MARCOS ANTONIO RAFAEL DA FONSECA -				
1 2 3 4 5 6 7 8 9 10				

Figura 127 Seleção de registros em lote

Para construir esse exemplo, a datagrid terá uma ação de seleção de linha, que por sua vez, marcará ou desmarcará o registro clicado em um vetor na sessão. Este mesmo vetor será utilizado como base em uma função de formatação para decidir quais as linhas dadatagrid serão destacadas com uma cor diferente. A partir do momento em que temos estes dados em sessão, podemos realizar qualquer ação sobre estes.

Para construir nosso exemplo, utilizaremos o Trait `AdiantiStandardListTrait`. Assim, não precisaremos nos preocupar com ações básicas como o carregamento de registros para adatagrid. Em seu método construtor realizamos algumas definições básicas como o banco de dados em uso (`setDatabase`), a classe a ser manipulada (`setActiveRecord`), e ordem default (`setDefaultOrder`). Estes parâmetros são utilizados para o carregamento efetivo dadatagrid.

Em seguida, criamos um formulário (`$this->form`), que será utilizado para a localização e filtro de registros nadatagrid. Este formulário conterá um campo "`description`", uma ação de buscas "Find", vinculada ao método `onSearch()`, fornecido pelo Trait.

Adatagrid, bem como seus campos, são criados em seguida. Neste exemplo, utilizaremos uma função de transformação sobre a coluna "`id`", chamada `formatRow()`. O papel desta função, que será detalhado mais adiante, será de "destacar" com uma cor diferente, os registros selecionados, que serão identificados pelo vetor em sessão.

Em seguida, criamos a ação (`$action1`) que será responsável por marcar ou desmarcar os registros dadatagrid. Ela estará vinculada ao método `onSelect()`. Então, sempre que uma linha for clicada, este método será executado. Basicamente ele será responsável por manter uma variável de sessão com os registros selecionados.

Ao final do método construtor, a paginação é criada e os objetos são empacotados em um `TPanelGroup`. Neste painel, adicionaremos uma ação para exibir os resultados selecionados pelo usuário. Esta ação estará vinculada ao método `showResults()`, que abrirá uma janela para exibir os registros.

`app/control/Organization/ComplexViews/ProductSelectionList.class.php`

```
<?php
class ProductSelectionList extends TPage
{
    protected $form;
    protected $datagrid;
    protected $pageNavigation;

    // importa Trait com onReload, onSearch, onDelete...
    use Adianti\Base\AdiantiStandardListTrait;

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

        // realiza definições básicas
        $this->setDatabase('samples'); // banco
        $this->setActiveRecord('Product'); // classe
        $this->setDefaultOrder('id', 'asc'); // ordem
```

```

// define filtro de busca (campo do banco, operador, campo do form)
$this->addFilterField('description', 'like', 'description');
$this->setLimit(10);

// cria o formulário
$this->form = new BootstrapFormBuilder('form_search_Product');
$this->form->setFormTitle(_t('Batch selection list'));

// cria o campo de buscas
$description = new TEntry('description');
$this->form->addField([new TLabel('Description')], [$description] );

// mantém o formulário preenchido com os dados da última busca
$this->form->setData( TSession::getValue('ProductSelectionList_filter_data') );

// adiciona ação de busca
$this->form->addAction('Find', new TAction([$this, 'onSearch']), 'fa:search');

// cria adatagrid
$this->datagrid = new BootstrapDatagridWrapper(new TDataGrid);
$this->datagrid->style = 'width: 100%';

// cria as colunas dadatagrid
$column_id = new TDataGridColumn('id', 'Id', 'left');
$column_description = new TDataGridColumn('description', 'Description', 'left');
$column_sale_price = new TDataGridColumn('sale_price', 'Sale Price', 'left');

// adiciona as colunas dadatagrid
$this->datagrid->addColumn($column_id);
$this->datagrid->addColumn($column_description);
$this->datagrid->addColumn($column_sale_price);

// cria um transformer sobre a coluna de id
$column_id->setTransformer([$this, 'formatRow'] );

// cria ação de marca/desmarca registro
$action1 = new TDataGridAction([$this, 'onSelect'],
    ['id' => '{id}',
     'register_state' => 'false']);
$action1->setButtonClass('btn btn-default');

// adiciona a ação nadatagrid
$this->datagrid->addAction($action1, 'Select', 'fa:square-o fa-fw black');

// cria estrutura dadatagrid em memória
$this->datagrid->createModel();

// cria o paginador
$this->pageNavigation = new TPageNavigation;
$this->pageNavigation->setAction(new TAction([$this, 'onReload']));

// empacota adatagrid em um painel
$panel = new TPanelGroup;
$panel->add($this->datagrid);
$panel->addFooter($this->pageNavigation);
$panel->addHeaderActionLink('Show results', new TAction([$this, 'showResults']), 'fa:check-circle-o' );

// empacota verticalmente formulário, edatagrid
$container = new TVBox;
$container->style = 'width: 100%';
$container->add(new TXMLBreadCrumb('menu.xml', __CLASS__));
$container->add($this->form);
$container->add($panel);

parent::add($container);
}

```

O método `onSelect()` será acionado sempre que o usuário clicar sobre uma linha da datagrid. Basicamente, ele receberá na posição `$param['id']` o ID do registro, sempre que o usuário clicar sobre o mesmo. Em primeiro lugar, este método buscará de uma variável de sessão os objetos já selecionados (`$selected_objects`). Em seguida, fará um teste: se o objeto já se encontra no vetor de sessão (`if isset`), então, a posição correspondente ao seu ID é limpa (`unset`); caso contrário, é armazenada na posição correspondente ao ID (`$selected_objects[$object->id]`), todo o objeto na forma de Array (`$object->toArray()`). Assim, todos dados necessários ficam disponíveis na sessão. Após marcarmos ou desmarcarmos a posição correspondente no vetor, jogamos a variável de volta à sessão (`setValue`) e recarregamos a datagrid por meio do método `onReload()`, que já refletirá a nova marcação.

```
public function onSelect($param)
{
    // lê os objetos de uma variável de sessão
    $selected_objects = TSession::getValue('__CLASS__._selected_objects');

    TTransaction::open('samples');
    $object = new Product($param['id']); // carrega o objeto
    if (isset($selected_objects[$object->id]))
    {
        unset($selected_objects[$object->id]);
    }
    else
    {
        $selected_objects[$object->id] = $object->toArray();
    }
    TSession::setValue('__CLASS__._selected_objects', $selected_objects);
    TTransaction::close();

    // recarregadatagrid
    $this->onReload( func_get_arg(0) );
}
```

Não basta que a ação `onSelect()` marque/desmarque os objetos em sessão. É necessário que alguma função crie um destaque visual para que o usuário saiba quais registros estão marcados. Este é o papel do método `formatRow()`, definido como transformer da coluna `id`. O objetivo de um transformer é modificar uma determinada coluna. Um transformer é aplicado sobre todas as linhas da datagrid. Por padrão, o transformer recebe como parâmetros: o valor atual da coluna (`$value`), o objeto de dados completo (`$object`), e a linha criada para o HTML (`$row`), podendo utilizar quaisquer uma destas variáveis. Neste caso, utilizaremos a variável `$value`, já que esta conterá o `id` atual do registro da datagrid, e utilizaremos também a variável `$row`, que representa a linha criada no HTML, para alterar sua cor de fundo por meio da definição de estilo (`style`).

Este método lê a variável de sessão com os id's selecionados, e verifica se o id atual (`$value`) está nesta lista. Caso esteja, a linha corrente tem seu fundo preenchido com uma cor (`$row->style`). Além disso, é utilizado o método `find()` para localizar o primeiro ícone (`i`) da linha, que na verdade é o ícone da ação. Ao encontrá-lo, o ícone é alterado para o ícone de checkbox preenchido (`check-square-o`), para dar o destaque necessário representando que aquele registro está marcado.

```

public function formatRow($value, $object, $row)
{
    $selected_objects = TSession::getValue(__CLASS__.'._selected_objects');

    if ($selected_objects) {
        if (in_array( (int) $value, array_keys( $selected_objects ) ) )
        {
            $row->style = "background: #abdef9";
            $button = $row->find('i', ['class'=>'fa fa-square-o fa-fw black'])[0];
            if ($button)
            {
                $button->class = 'fa fa-check-square-o fa-fw black';
            }
        }
    }

    return $value;
}

```

O método `showResults()` é acionado por meio de um botão no topo do painel (`TPanelGroup`) e tem como objetivo apresentar os registros selecionados. Sua tarefa é relativamente simples, pois os registros selecionados estão prontos na variável de sessão. Então, este método simplesmente cria umadatagrid (`TDataGrid`), usando o wrapper para Bootstrap. Em seguida, lê o vetor da variável de sessão (`TSession::getValue()`), e então, percorre estes objetos, adicionando-os àdatagrid, por meio do método `addItem()`. Por fim, criamos uma janela por meio do método `TWindow::create()`, adatagrid é adicionada à janela pelo método `add()`, e a janela é exibida pelo método `show()`.

```

public function showResults()
{
    $datagrid = new BootstrapDatagridWrapper(new TDataGrid);
    $datagrid->width = '100%';

    $datagrid->addColumn( new TDataGridColumn('id', 'ID', 'left') );
    $datagrid->addColumn( new TDataGridColumn('description', 'Description',
    'left') );
    $datagrid->addColumn( new TDataGridColumn('sale_price', 'Sale Price', 'right')
);

    // cria a estrutura dadatagrid
    $datagrid->createModel();

    $selected_objects = TSession::getValue(__CLASS__.'._selected_objects');
    ksort($selected_objects);
    if ($selected_objects) {
        $datagrid->clear();
        foreach ($selected_objects as $selected_object)
        {
            $datagrid->addItem( (object) $selected_object );
        }
    }

    $win = TWindow::create('Results', 0.6, 0.6);
    $win->add($datagrid);
    $win->show();
}

```

CAPÍTULO 6

Template para criação de sistemas

No início, o Adianti Framework era nada mais do que um Framework puro, sendo que cada usuário implementava de sua maneira o controle de login, de permissões de acesso, dentre outros. Até que resolvemos criar o que chamamos de “Template” para fornecer um gabarito padrão para nossos usuários criarem aplicações. O Template permite ao desenvolvedor definir as opções do menu, cadastrar os programas, usuários, grupos, permissões de acesso, sendo que o Template cuida da correta montagem de menus e controle de acesso. Além disso, o Template registra logs de acesso, de alteração de registros, de SQL, e permite aos usuários compartilharem documentos, trocarem mensagens, e receberem notificações do sistema, de uma maneira padronizada.

6.1 Visão geral

Nas primeiras versões do Framework, não havia uma aplicação que demonstrasse, de maneira completa, como implementar um controle de permissões, o que levou diferentes desenvolvedores a criarem mecanismos próprios para esta funcionalidade.

Após um tempo, percebemos que uma necessidade comum devia ser implementada de maneira padronizada, para aumentar a qualidade das aplicações. Dessa maneira criamos o Template, um gabarito para criação de novas aplicações com o Adianti Framework. A cada nova versão, esse gabarito ganha novas funcionalidades. Inicialmente ele oferecia controle de permissões de acesso, por meio do cadastro de usuários, grupos, e programas. Posteriormente foram agregadas funcionalidades de registro de logs de acesso, de alteração de registros e log de SQL, e em seguida de comunicação entre usuários por meio de troca de mensagens e compartilhamento de documentos.

Obs: O Template pode ser baixado em <http://www.adianti.com.br/framework-template>.

6.1.1 Formulário de Login

Na medida em que o usuário acessar o sistema pela primeira vez, terá acesso ao formulário de login. Basicamente o formulário de login contém dois campos obrigatórios: usuário e senha. No entanto, também podemos habilitar que o usuário preencha a unidade (filial, empresa) e também o idioma. Estes duas últimas são totalmente opcionais e habilitadas no `application.ini`. Em seções posteriores, abordaremos este arquivo. Podemos utilizar a informação da unidade para representar o acesso à uma determinada empresa/filial, e segmentar os dados a serem visualizados. Já a combo de idioma pode ser utilizada para que o próprio usuário escolha o idioma que deseja utilizar no sistema.

A partir do login, é identificado o usuário e são carregadas suas permissões de acesso com base no cadastro de permissões. Assim, o usuário logado somente visualizará no menu do sistema as opções que possui acesso. Mesmo que este usuário tente acessar tais programas pela URL, não conseguirá pois o sistema de permissões não permitirá.

Os links [[Criar conta](#)] e [[Redefinir senha](#)] são opcionais e podem ser desabilitados a qualquer momento mediante arquivo de configuração `application.ini`. Com o link Criar conta, podemos permitir que os usuários do sistema realizem um auto cadastro. Já o botão Redefinir senha, permite ao usuário final criar uma nova senha, caso tenha esquecido a sua.

The screenshot shows the 'LOG IN' page of the Adianti Framework ERP Template III. At the top, there is a header bar with the text 'Adianti Framework ERP Template III'. Below the header, the page title 'MARCOS ANTONIO RAFAEL DA FONSECA -' is displayed. The main content area has a blue header 'LOG IN'. It contains four input fields: 'admin' for the user, 'Senha' (password) which is currently empty, 'Unit A' for the unit, and 'Português' for the language. Below these fields is a large blue 'Entrar' (Enter) button. At the bottom of the page, the text 'MARCOS ANTONIO RAFAEL DA FONSECA -' is repeated, followed by two links: 'Criar conta' and 'Redefinir senha'.

Figura 128 Formulário de login

6.1.2 Módulos

Nesta seção, vamos conhecer os principais módulos (conjuntos de funcionalidades) que acompanham o Template. O primeiro módulo, que é essencial para o funcionamento do Template, é o módulo “Administração”. Este módulo é essencial pois controla o acesso ao sistema por meio de login, e o controle de acesso a cada programa.

O módulo “Administração” registra as informações em uma base de dados própria, que pode ser separada da aplicação ou na mesma base, conforme a necessidade. Esta base de dados é configurada pelo arquivo `app/config/permission.ini`, visto mais adiante. Por padrão, o Template sai de fábrica com as permissões armazenadas em SQLite. É possível armazenar as permissões de acesso em outros bancos, tais como MySQL, ou PostgreSQL. Também é possível armazenar estes dados na mesma base de dados da aplicação, bastando alterar os apontamentos do `permission.ini`.

Este módulo engloba funcionalidades como: cadastro de programas, cadastro de grupos de usuários, cadastro de unidades, cadastro de usuários, um navegador de base de dados, um painel para executar comandos SQL, consulta do PHP Info, painel para demonstrar os módulos habilitados do PHP, e formulário para definição de preferências do sistema, conforme pode ser visto na figura a seguir.

	Id	Nome	Login	Email	Ativo
	1	Administrator	admin	admin@admin.net	
	2	User	user	user@user.net	

Figura 129 Módulo de Administração

Obs: O visual do Template pode ser customizado. Além de podermos configurar as cores do tema utilizado, existem outros temas disponíveis, como um utilizando Material Design.

Outro módulo bastante importante é o módulo de registro e consulta a logs do sistema. Este módulo permite o registro e a consulta de logs de acesso, logs de alteração de registros, logs de SQL, e logs de PHP (erros e warnings gerados).

O módulo de logs registra as informações em uma base de dados própria para que não consuma espaço da base de dados da aplicação, o que é configurado pelo arquivo `app/config/log.ini`. Por padrão, o Template sai de fábrica armazenando os logs em SQLite, mas você pode mudar isso no `log.ini`, armazenando em MySQL, PostgreSQL, dentre outros. É importante não armazenar os logs juntamente com a aplicação, em função do espaço que os dados ocuparão, prejudicando o backup. Os logs podem ser habilitados conforme a necessidade. Na próxima figura, temos o módulo de logs.

The screenshot shows the 'Access Log' section of the 'Logs' module. The left sidebar has a dark theme with a user icon and the name 'MARCOS ANTONIO RAFAEL DA FONSECA'. The main area has a light blue header with the title 'LOGS > LOG DE ACESSO'. Below it is a search bar with placeholder 'Buscar' and a 'Logins' dropdown. A 'Buscar' button is at the bottom of the search area. The main content is a table titled 'Access Log' with columns: id, sessionid, Login, login_time, logout_time, and IP. The table contains 8 rows of log entries. At the bottom right of the table, there is a message: 'MARCOS ANTONIO RAFAEL DA FONSECA'.

	id	sessionid	Login	login_time	logout_time	IP
	12	nd2j502as4bn6ln5ilb6gfb6	admin	2019-10-07 21:05:19		127.0.0.1
	11	glos0lhos5kusmo650k5cf9bf5	admin	2019-10-07 19:57:10		127.0.0.1
	10	2m3i82k9pfim7spag0pj8o9	admin	2019-10-07 19:44:36	2019-10-07 19:47:19	127.0.0.1
	9	124v8lhjholt2725t27se8409	admin	2019-09-28 20:43:21		127.0.0.1
	8	i9opiuipjd55vtcknvcjtstd3im	admin	2019-09-28 14:44:58		127.0.0.1
	7	0160shh8k9tdel2gb50632ikm	admin	2019-09-27 22:11:36		127.0.0.1
	6	3dptkmjdjvouu10ngao0knroe0	admin	2019-09-27 20:24:18		127.0.0.1
	5	n7po44hsf32in6b2nsbhfcqvq	admin	2019-09-21 14:42:24	2019-09-21 14:42:28	127.0.0.1

Figura 130 Módulo Logs

Obs: Logs de SQL e de alterações de registros precisam ser habilitados. Logs de SQL são habilitados em nível de base de dados, já logs de alteração de registros são habilitados em nível de tabela. Na seção posterior “Módulo Logs”, abordaremos como os diferentes tipos de logs podem ser habilitados para uma determinada aplicação.

O próximo módulo que o Template oferece é o módulo de comunicação. Este módulo oferece recursos de envio e compartilhamento de documentos entre usuários e grupos, bem como recursos de comunicação entre usuários (inbox).

Os usuários podem submeter arquivos (menu Documentos → Enviar documentos) e compartilhar estes com outros usuários do sistema. Já as mensagens são trocadas por meio do ícone em formato de envelope, presente no canto superior direito.

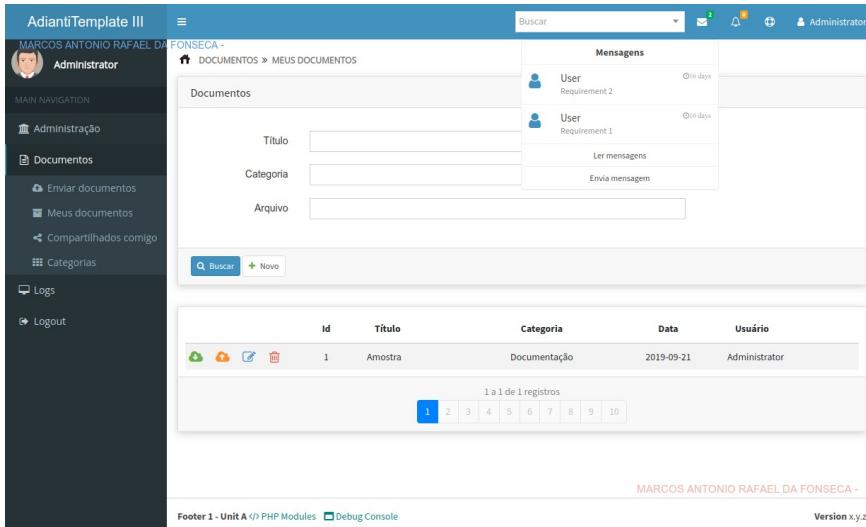


Figura 131 Módulo comunicação

6.1.3 Estrutura de diretórios

O Template é um gabarito para criação de aplicações que utilizam o Adianti Framework. Além do próprio framework, o template entrega o seguinte conteúdo, além de versões modificadas de arquivos como `index.php` e `engine.php`.

Tabela 5. Conteúdos do Template

app	Diretório da aplicação
config application.ini permission.ini log.ini communication.ini	Pasta com configurações Configuração global da aplicação Configuração de acesso para a base de permissões Configuração de acesso para a base de logs Configuração de acesso para a base de comunicação
control admin log communication public	Classes de controle Classes para administração (permissões, preferências) Classes para gestão de logs Classes para comunicação (documentos, mensagens) Classes de acesso público
database permission.db log.db communication.db	Arquivos de base de dados em formato SQLite Base de dados em SQLite para permissões Base de dados em SQLite para logs Base de dados em SQLite para comunicação
lib barcode html include menu pdf reports util validator widget	Bibliotecas da Aplicação Classes para geração de códigos de barras Classes para geração de documentos Bibliotecas Javascript necessárias (.js) Classe para geração de menu conforme o tema Classe para geração de PDF Classes para escrita de relatórios em tabelas Classes utilitárias (tradução, rotas, e-mails) Validadores de formulários específicos da aplicação Componentes (widgets) específicos da aplicação

model	Classes de modelo (Active Records) Classes de modelo de administração Classes de modelo de logs Classes de modelo de comunicação
service	Classes de serviço Serviços de autenticação Serviços de log Serviços de REST Serviços do sistema
templates	Diferentes temas para a aplicação Tema 3, baseado no AdminLTE (Bootstrap) Tema 4, baseado no AdminBSB (Material Design)

6.1.4 O application.ini

É no arquivo `app/config/application.ini` que são realizadas as configurações globais da aplicação. Como pode ser visto a seguir, este arquivo possui a definição de `timezone` padrão, o que afeta cálculos de tempo. Também temos o idioma padrão no campo `language`. Já no campo `application`, definimos o nome da aplicação. É muito importante você preencher este campo, visto que ele é usado para separar sessões de aplicações. Assim, se você tiver duas aplicações rodando no mesmo domínio, e com o mesmo `application`, ao logar em uma estará automaticamente logado na segunda, se não utilizar nomes diferentes em `application`.

A variável `theme` define o tema da aplicação (diretório em `app/templates`). A variável `debug` deve ser usada em desenvolvimento. Quando ligada, as mensagens de exceção são mais completas (contendo nomes de arquivos e linhas de erro). A variável `multilang` liga o modo multi-idioma. Neste caso, o próprio usuário poderá escolher o idioma na tela de login. As opções são disponibilizadas em `lang_options`.

A variável `user_register` exibe o link [[Criar conta](#)] na tela de login, permitindo que usuários se auto cadastram na aplicação, `reset_password` libera o link [[Redefinir senha](#)] também na tela de login da aplicação, permitindo que usuários façam nova definição de senha, `default_groups` define um ou vários códigos de grupos (separados por vírgula) que serão atribuídos para usuários que se cadastrarem pelo link [[Criar conta](#)]; e `default_screen`, que define o código do programa inicial para usuários que se auto cadastrarem na aplicação.

`app/config/application.ini`

```
[general]
timezone = America/Sao_Paulo
language = pt
application = template
theme = theme3
debug = 1
multi_lang = "1"
lang_options[pt] = Português
lang_options[en] = English
lang_options[es] = Español
```

```
[permission]
user_register = "1" ; users may auto register
reset_password = "1" ; users may reset password
default_groups = "2" ; default groups for auto registered users
default_screen = "10"
```

Obs: A opção de redefinição de senha envia por e-mail para a conta pré-cadastrada do usuário um link com um token JWT, uma abordagem comprovadamente segura. Para funcionar, é necessário que um servidor de e-mails esteja corretamente configurado nas preferências do sistema, que são acessadas pelo módulo Administração.

6.1.5 Menu da aplicação

O menu da aplicação é montado dinamicamente por meio de um arquivo XML. Este arquivo XML deverá conter toda a estrutura de menu do sistema, com todas as opções que possam estar disponíveis para os usuários. A partir do login, o Template selecionará, deste arquivo, somente as opções que o usuário possui permissão de acesso. Na figura a seguir, podemos ver um exemplo deste menu no tema 4 (Material Design).

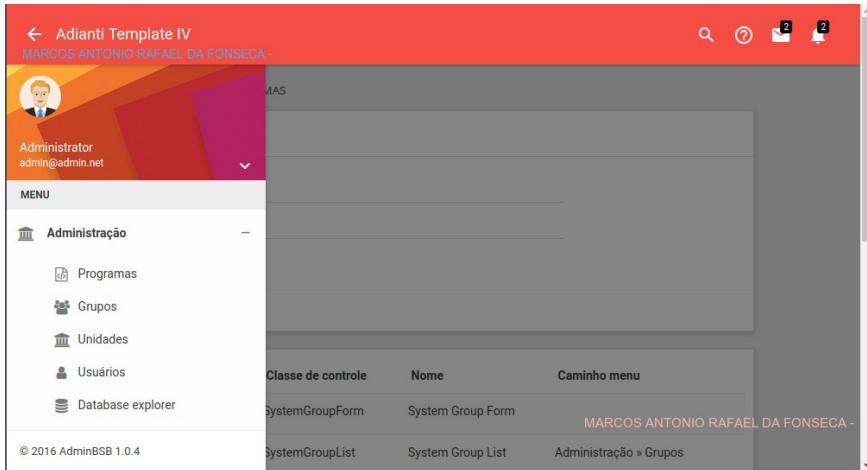


Figura 132 Menu dinâmico da aplicação

A seguir, temos o menu do Template, que possui a mesma estrutura que o menu já apresentado no livro, quando referenciamos a aplicação Tutor. Neste exemplo, foram colocadas somente algumas opções, como as administrativas. A simbologia `_t{}` é utilizada para realização de tradução de palavra, conforme o dicionário da aplicação.

É importante notar que o usuário somente visualizará no menu e terá acesso aos programas que o seu perfil de acesso (cadastro de usuários e grupos) permitir. Estes cadastros administrativos serão vistos mais adiante.

menu.xml

```

<menu>
    <menuitem label='_t{Administration}'>
        <icon>fa:university fa-fw</icon>
        <menu>
            <menuitem label='_t{Programs}'>
                <icon>fa:file-code-o fa-fw</icon>
                <action>SystemProgramList</action>
            </menuitem>
            <menuitem label='_t{Groups}'>
                <icon>fa:users fa-fw</icon>
                <action>SystemGroupList</action>
            </menuitem>
            <menuitem label='_t{Units}'>
                <icon>fa:university fa-fw</icon>
                <action>SystemUnitList</action>
            </menuitem>
            <menuitem label='_t{Users}'>
                <icon>fa:user fa-fw</icon>
                <action>SystemUserList</action>
            </menuitem>
            <menuitem label='_t{Database explorer}'>
                <icon>fa:database fa-fw</icon>
                <action>SystemDatabaseExplorer</action>
            </menuitem>
        </menu>
    </menuitem>

    <menuitem label='Logout'>
        <menu>
            <menuitem label='Logout'>
                <icon>fa:sign-out fa-fw</icon>
                <action>LoginForm#method=onLogout#static=1</action>
            </menuitem>
        </menu>
    </menuitem>
</menu>
```

Obs: As opções relativas aos programas criados pelo desenvolvedor, devem ser adicionadas em blocos `<menu> <menuitem>` após as opções já existentes relativas ao sistema (módulos administração, logs, e comunicação).

6.1.6 Layout e temas

A aplicação possui mais de um arquivo de layout. O primeiro (`layout.html`), é similar ao layout do Tutor e será utilizado quando o usuário estiver logado. Já o layout de login (`login.html`) é um arquivo utilizado somente para a tela de login. No código a seguir, estão em destaque, as marcações `{LIBRARIES}` e `{HEAD}` que são marcações que devem estar presentes no layout, e que são automaticamente substituídas pelas bibliotecas do Framework, a DIV `adianti_div_content`, que é a DIV onde o conteúdo das páginas será renderizado, `adianti_online_content` que é a DIV onde o conteúdo de novas janelas será exibido, e `adianti_online_content2` que é a DIV onde o conteúdo de janelas sobre janelas será renderizado, e `adianti_right_panel`, onde são abertas cortinas laterais. Não exclua estes blocos.

app/templates/themeX/login.html

```
<!DOCTYPE html>
<html>
    <head>
        <title>Adianti ERP Template</title>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        {LIBRARIES}
        {HEAD}
    </head>

    <body>
        <div class="adianti_container">
            <div class="header">
                ...
            </div>

            <div class="body">
                <div id="adianti_div_content" class="content"></div>
                <div id="adianti_online_content"></div>
                <div id="adianti_online_content2"></div>
                <div id="adianti_right_panel" class="right-panel"></div>
            </div>
            <div style="clear:both;"></div>
        </div>
    </body>
</html>
```

O lançamento do Template foi um grande sucesso e auxiliou muitos desenvolvedores do Framework a construírem aplicações com controle de permissões de acesso, bem como opções administrativas para gerenciamento das aplicações. Um pouco depois de seu lançamento, novas demandas começaram a surgir, como a criação de novos layouts baseados na biblioteca Bootstrap.

Para atender a essas demandas, foram criados novos “temas” para o Template. Os temas ficam armazenados na pasta `app/templates` (`theme1`, `theme2`, `theme3`, `theme4`). Para realizar a alteração do tema, basta mudar a definição no arquivo de configurações globais da aplicação: `application.ini`. Os temas 1 e 2 são antigos e não são mais mantidos.

app/config/application.ini

```
[general]
timezone = America/Sao_Paulo
language = pt
application = template
theme = theme3
```

A seguir, apresentaremos as características visuais dos temas distribuídos de maneira embarcada ao Template, e que podem ser alterados a qualquer momento pelo arquivo `application.ini` como apontado anteriormente.

Atualmente o Framework é distribuído com dois temas: O tema 3, baseado no template AdminLTE, e o tema 3, baseado no template AdminBSB. Ambos são baseados na biblioteca Bootstrap, mas o segundo segue as diretrizes do Material Design.

O Tema 3 é baseado em um template chamado “Admin LTE”, de grande sucesso, de uso livre e construído sobre a biblioteca Bootstrap. Totalmente responsivo, com menu lateral e opções de busca de programas, troca de mensagens, visualização de notificações e edita perfil do usuário na barra de topo do sistema.

Quando o espaço for muito reduzido, o menu lateral é transformado em um menu do tipo sanduíche, com recolhimento automático, liberando espaço na tela para o conteúdo a ser exibido (formulários, listagens, etc). A figura a seguir, demonstra o tema 3 em uso.

ID	Nome	Login	Email	Ativo
1	Administrator	admin	admin@admin.net	Sim
2	User	user	user@user.net	Sim

Figura 133 Tema 3 do Template

O último tema desenvolvido, e também de grande sucesso, é o Tema 4, baseado no Template livre chamado AdminBSB, que implementa os conceitos de Material Design. Neste tema, os formulários e datagrids também se ajustam ao tamanho da tela.

Este tema possui o menu de opções posicionado à esquerda, bem como opções de notificações e mensagens ao lado do menu do usuário, no canto superior direito. Quando o espaço for muito reduzido, o menu lateral também é transformado em um menu do tipo sanduíche, com recolhimento automático, liberando espaço na tela para o conteúdo a ser exibido (formulários, listagens, etc). A figura a seguir, demonstra o Tema 4 em uso.

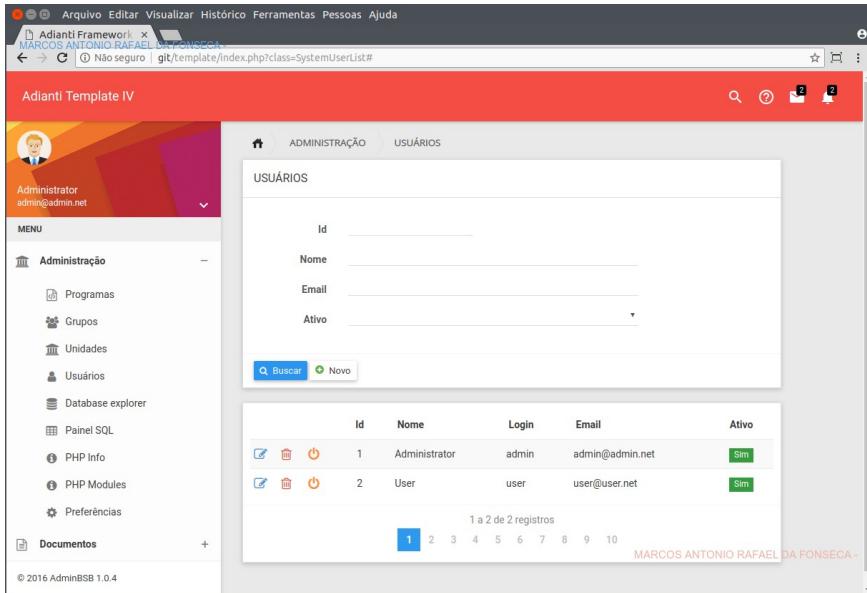


Figura 134 Tema 4 do Template

6.2 Módulo Administração

Nesta seção abordaremos o módulo Administração, sua modelagem e funcionalidades.

6.2.1 Diagrama de classes

Na próxima figura, temos o diagrama de classes do sistema, com as classes da camada de modelo relativas ao controle de permissões de acesso (`app/model/admin`). A seguir, temos uma lista com as classes Active Record:

- SystemUser**: representa um usuário do sistema. Um usuário poderá ter agregação com grupo (**SystemGroup**), e com programa (**SystemProgram**). Isto significa que um usuário poderá possuir vários grupos e vários programas. Além disso, usuário pode ter uma página inicial (*front page*), e estar associado com uma unidade organizacional (**SystemUnit**) principal, dentre várias outras;
- SystemGroup**: representa um grupo de usuários. Um grupo tem agregação com programa (**SystemProgram**). Isto significa que um grupo poderá possuir vários programas;
- SystemProgram**: representa um programa. Um programa possui um nome (**name**) e uma classe de controle (**controller**). O atributo **controller** deve ser preenchido com o nome da classe de controle, que desejamos dar acesso.
- SystemUnit**: representa uma unidade organizacional. Pode ser uma filial ou um departamento de uma organização.
- SystemPreference**: representa uma preferência do sistema. Possui chave (**id**) e valor (**value**). Podem existir muitas preferências, como e-mail, etc.

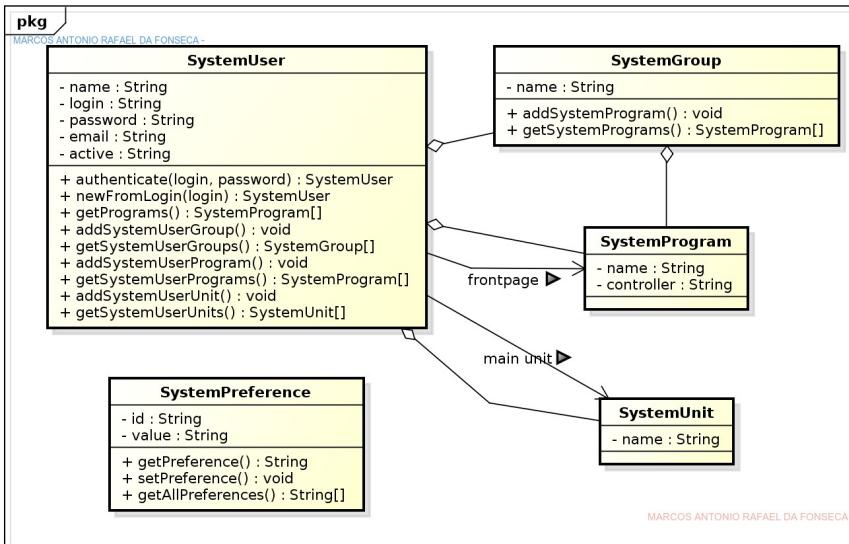


Figura 135 Diagrama de classes do módulo de permissões Template

Quando um usuário estiver vinculado a um grupo, automaticamente terá acesso a todos os programas daquele grupo. Este modelo permite que se concedam permissões de programas para grupos inteiros ou a usuários.

Quando os acessos aos programas forem somente por grupos, não é necessário conceder programas especificamente a um usuário (por meio da agregação `SystemUser` – `SystemProgram`). Mas esta agregação é útil quando desejamos conceder direitos específicos para um determinado usuário, além da permissão que o mesmo possui pelo grupo.

6.2.2 Modelo relacional

No modelo relacional do Template, temos as tabelas a seguir. Junto a cada tabela, temos a explicação de qual Active Record ela representa.

- `system_user`: armazena os usuários do sistema;
- `system_group`: armazena um grupo de usuários;
- `system_program`: armazena os programas, uma página é um registro;
- `system_user_group`: armazena a relação entre `SystemUser` e `SystemGroup`. Nesta tabela será armazenado cada vínculo de usuário a um grupo de usuários;
- `system_group_program`: armazena a relação de agregação entre `SystemGroup` e `SystemProgram`, ou seja, cada vínculo de um grupo a um programa;
- `system_user_program`: armazena a relação de agregação entre `SystemUser` e `SystemProgram`, ou seja, cada vínculo de um usuário a um programa;
- `system_unit`: armazena uma unidade organizacional (departamento, filial);
- `system_user_unit`: armazena o vínculo entre usuários e unidades;
- `system_preference`: armazena os registros (chave e valor) das preferências.

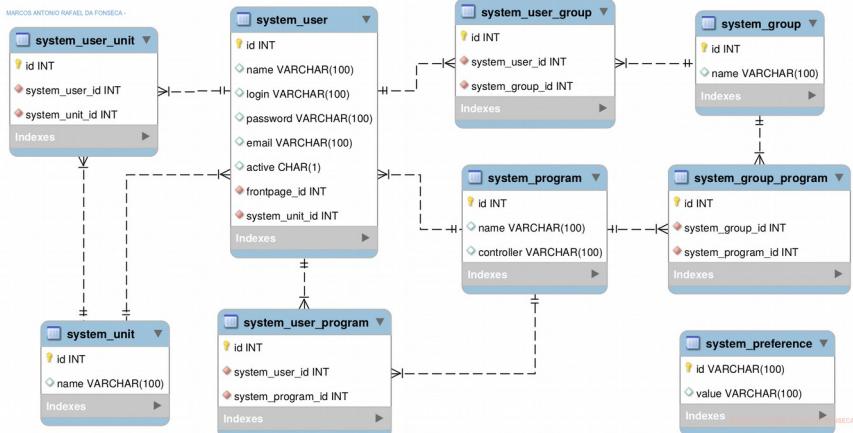


Figura 136 Modelo relacional do módulo Administração

O acesso ao banco de dados de permissões é realizado pelos arquivos:

- app/config/permission.ini**: Arquivo de configuração de acesso à base de dados. Contém as credenciais de acesso, localização do banco de dados, etc.
- app/database/permission.db**: Arquivo contendo as tabelas do módulo de administração no formato SQLite. É utilizado para instalação inicial;
- app/database/permission.sql**: Arquivo de criação da base de dados de administração. Pode ser criado em outros sistemas de banco de dados.

Caso você queira utilizar outro sistema de banco de dados para armazenar as tabelas do modelo de permissão, basta usar o script de criação (**permission.sql**) em outro banco de dados e alterar os dados de conexão dentro do arquivo **permission.ini**.

Obs: Não exclua sob hipótese alguma o arquivo **permission.ini**. Mesmo que você decida reunir a base de permissões com a base da aplicação, mantenha os diferentes INI's apontando para a mesma base de dados, uma vez que existem vários pontos dentro do Template que referenciam este arquivo no momento de abrir conexão com a base de dados.

6.2.3 Formulário de login

Na medida em que o usuário acessar o sistema pela primeira vez, terá acesso ao formulário de login. Nesta seção, será explicado tecnicamente o que ocorre na medida em que o usuário realiza o acesso ao sistema.

O formulário de login sempre solicitará usuário e senha. Opcionalmente, ele poderá solicitar unidade e idioma. Caso no `application.ini` o parâmetro `multiunit` esteja ligado (1), então o formulário de login solicitará que o usuário escolha em qual unidade ele quer logar. Uma variável de sessão com o código da unidade será alimentada no ato do login. Caso no `application.ini` o parâmetro `multi_lang` esteja ligado (1), então o formulário de login solicitará que o usuário escolha o idioma da interface, conforme as opções (`lang_options`) disponibilizadas no `application.ini`.

Adianti Framework ERP Template III

MARCOS ANTONIO RAFAEL DA FONSECA -

LOG IN

	admin
	Senha
	Unit A
	Português

Entrar

MARCOS ANTONIO RAFAEL DA FONSECA -

[Criar conta](#) [Redefinir senha](#)

Figura 137 Formulário de login

O formulário de login é representado pela classe `LoginForm`. Para simplificar a explicação, neste caso vamos suprimir o método construtor, onde a interface é construída, uma vez que ela é bastante simples, e possui somente a definição dos campos (login, senha, unidade, e idioma) além da uma ação de login. Assim, vamos nos concentrar no método `onLogin()`, que é executado sempre que o usuário clicar no botão “Entrar”.

O método `onLogin()` é responsável por realizar validação de preenchimento de campos, executar a autenticação do usuário, e também alimentar variáveis de sessão.

O método `onLogin()` inicialmente valida os campos preenchidos, como login, senha e unidade (caso esteja ligado o parâmetro `multiunit`). Caso no `application.ini` o parâmetro `multiunit` (multi unidade) esteja ligado, então o usuário deverá obrigatoriamente selecionar uma unidade dentre aquelas cadastradas.

Após as validações, a sessão é regerada por meio do método `TSession::regenerate()` e o usuário é autenticado pelo `ApplicationAuthenticationService::authenticate()`. Este método usa o autenticador nativo do sistema, mas é possível definir um método alternativo de autenticação (será visto em um dos próximos tópicos). O método `authenticate()` valida o login e senha do usuário. Caso login e senha estejam corretos, o usuário (objeto `SystemUser`) é retornado, caso contrário, uma exceção é lançada, interrompendo o fluxo de execução do bloco `try` e exibindo a exceção no `catch`.

Após autenticado, o método `ApplicationAuthenticationService::setUnit()` é executado. Este método verifica se o parâmetro `multiunit` (multi unidade) está ligado no `application.ini`. Se ligado, este método alimenta uma variável de sessão com a unidade selecionada, para que essa possa ser utilizada em filtros da aplicação.

Em seguida, é executado o método `ApplicationAuthenticationService::setLang()`. Este método verifica se o parâmetro `multi_lang` (multi idioma) está ligado no `application.ini`. Se ligado, este método alimenta uma variável de sessão com o idioma selecionado, que por sua vez é utilizada para tradução dos termos da aplicação.

Em seguida, é acionado o método `SystemAccessLog::registerLogin()`, que registra o acesso do usuário (será abordado mais adiante na seção sobre registro de logs). Por fim, caso o usuário tenha uma página de entrada (Front page) cadastrada, ele será direcionado para esta página, após o login, por meio do método `AdiantiCoreApplication::gotoPage()`, caso contrário, será direcionado para uma página vazia (`EmptyPage`).

app/control/admin/LoginForm.class.php

```
<?php
class LoginForm extends TPage
{
    protected $form;

    public static function onLogin($param)
    {
        $ini = AdiantiApplicationConfig::get();

        try {
            $data = (object) $param;

            // valida campos obrigatórios
            (new TRequiredValidator)->validate( _t('Login'), $data->login);
            (new TRequiredValidator)->validate( _t('Password'), $data->password);

            // valida preenchimento da unidade, se multi unidade ligado
            if (!empty($ini['general']['multiunit']) and
                $ini['general']['multiunit'] == '1')
            {
                (new TRequiredValidator)->validate( _t('Unit'), $data->unit_id);
            }
        }
    }
}
```

```

    // gera nova session
    TSession::regenerate();

    // autentica o usuário e senhas
    $user = ApplicationAuthenticationService::authenticate( $data->login,
                                                            $data->password );
    if ($user)
    {
        // define a unidade, se aplicável (multi unidade ligado)
        ApplicationAuthenticationService::setUnit( $data->unit_id ?? null );

        // define o idioma, se aplicável (multi idioma ligado)
        ApplicationAuthenticationService::setLang( $data->lang_id ?? null );

        // registra o acesso nos logs
        SystemAccessLogService::registerLogin();

        // carrega página inicial cadastrada no perfil do usuário
        $frontpage = $user->frontpage;
        if ($frontpage instanceof SystemProgram and $frontpage->controller)
        {
            AdiantiCoreApplication::gotoPage($frontpage->controller); // reload
            TSession::setValue('frontpage', $frontpage->controller);
        }
        else
        {
            AdiantiCoreApplication::gotoPage('EmptyPage'); // reload
            TSession::setValue('frontpage', 'EmptyPage');
        }
    }
    TTransaction::close();
}
catch (Exception $e)
{
    new TMessage('error', $e->getMessage());
    TTransaction::rollback();
}
}

```

O método `onLogout()` efetua o logout. Basicamente, a chamada ao método `SystemAccessLog::registerLogout()`, registra o log de saída, e em seguida a sessão é limpa (`freeSession`), e o usuário é transferido para a tela de login, por meio do `gotoPage()`.

```

function onLogout()
{
    SystemAccessLog::registerLogout();
    TSession::freeSession();
    AdiantiCoreApplication::gotoPage('LoginForm', '');
}
}

```

Para acessar o Template corretamente, é necessário conhecer antecipadamente os logins pré-cadastrados. Nesta tabela, temos os logins, um para cada papel de usuário.

Tabela 6. Logins e senhas para o template

Usuário	Login	Senha
Administrator	admin	admin
User	user	user

Após login, as seguintes variáveis de sessão alimentadas. Você pode utilizá-las em qualquer lugar da aplicação pelo método `TSession::getValue()`.

Tabela 7. Variáveis de sessão alimentadas no login

Variável	Tipo	Descrição
logged	boolean	TRUE se o usuário está logado
login	string	Login do usuário logado.
userid	string	ID do usuário logado.
usergroupids	array	Vetor com ID's dos grupos do usuário logado.
userunitids	array	Vetor com ID's das unidades do usuário logado.
username	string	Nome do usuário logado.
usermail	string	E-mail do usuário logado.
frontpage	string	Classe de entrada do usuário logado.
programs	array	Vetor dos programas permitidos para o usuário.
Opcionais (dependem de configurações que podem estar ou não ligadas)		
userunitid	string	Id da unidade selecionada pelo usuário no login (somente se o multi-unidade estiver ligado)
userunitname	string	Nome da unidade selecionada pelo usuário no login (somente se o multi-unidade estiver ligado)
unit_database	string	Base de dados vinculada à unidade selecionada no login (somente se o multi-database estiver ligado).
user_language	string	Idioma selecionado pelo usuário na tela de login (somente se o multi-language estiver ligado).

6.2.4 Multi unidade

O Template oferece a possibilidade da seleção de unidade na tela de login. O conceito de unidade pode ser utilizado para segmentação de empresas, filiais, departamentos ou conceitos relacionados.

Para que o Template solicite que o usuário escolha a unidade na tela de login, basta ligar o parâmetro `multiunit` no `application.ini`. Neste caso, o formulário de login exigirá que o usuário selecione uma das unidades vinculadas ao seu cadastro. Antes, é necessário cadastrar unidades e vinculá-las ao usuário. Somente serão listadas unidades vinculadas ao cadastro do usuário. Veja a seguir, como habilitar a solicitação de unidade.

`app/config/application.ini`

```
[general]
multiunit = 1
```

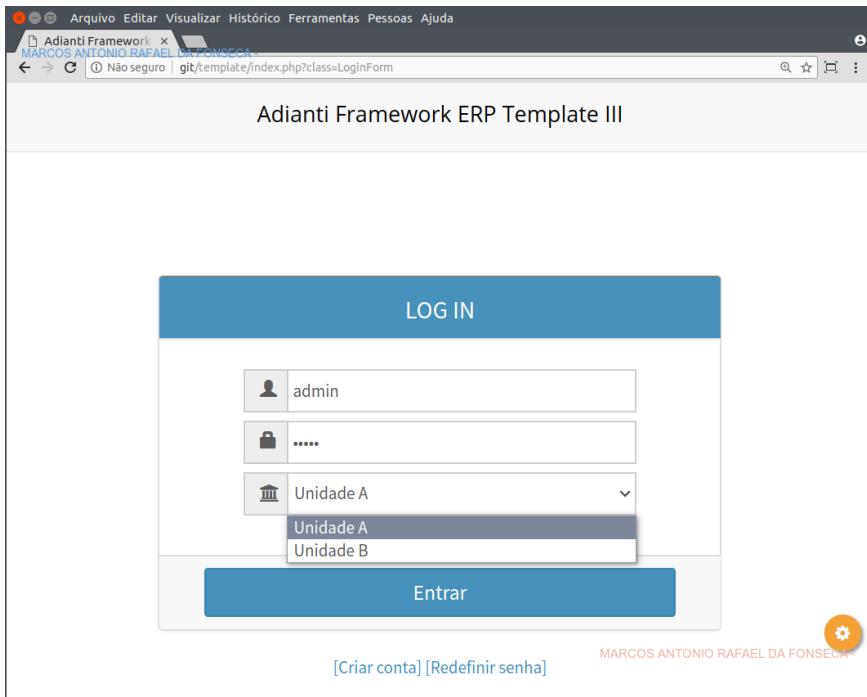


Figura 138 Seleção de unidade no formulário de Login

Após o login, as seguintes variáveis de sessão ficarão disponíveis na aplicação para uso.

Tabela 8. Variáveis de sessão para multi unidade

Variável	Tipo	Descrição
userunitid	string	Id da unidade selecionada pelo usuário no login (somente se o multi-unidade estiver ligado)
userunitname	string	Nome da unidade selecionada pelo usuário no login (somente se o multi-unidade estiver ligado)

A partir de qualquer ponto da aplicação, você poderá utilizar esta variável de sessão contendo o código da unidade selecionada na tela de login.

TSession::getValue('userunitid')

Um exemplo de utilização é ao filtrar os registros de uma datagrid, trazendo somente os registros de uma determinada unidade selecionada na tela de login:

```
$criteria = new TCriteria;
$criteria->add(new TFilter('unit_id', '=', TSession::getValue('userunitid')));
$this->setCriteria($criteria);
```

6.2.5 Multi database

Quando o multi unidade é ligado, também podemos ligar o `multi_database`. O multi database permite vincularmos bases de dados ao cadastro de unidades. Quando ligado, ao cadastrarmos uma unidade, também seremos solicitados a vincular um conector de base de dados à unidade. Para ligar o `multi_database`, basta alterar o `application.ini`.

`app/config/application.ini`

```
[general]
multi_database = "1"
```

A figura a seguir exibe o cadastro de unidades quando o `multi_database` está ligado.

Figura 139 Vínculo de conexão de base de dados na Unidade

Quando o multi database está ligado, a partir do login, a seguinte variável de sessão fica disponível para uso na aplicação.

Tabela 9 Variáveis de sessão para multi database

Variável	Tipo	Descrição
<code>unit_database</code>	string	Base de dados vinculada à unidade selecionada no login (somente se o multi-database estiver ligado).

A partir de qualquer ponto da aplicação, você poderá utilizar esta variável de sessão contendo o nome da conexão de banco de dados vinculada à unidade selecionada na tela de login.

```
TSession::getValue('unit_database')
```

Um exemplo de utilização é ao abrir uma conexão com a base de dados. Neste caso, podemos abrir uma conexão dinamicamente, com base no conector vinculado à unidade selecionada na tela de login.

```
TTransaction::open( TSession::getValue('unit_database') );
```

6.2.6 Internacionalização

Uma das preocupações ao construirmos aplicações globais é a internacionalização. O objetivo é escrevermos a aplicação somente uma vez e poder rodá-la em diversos idiomas. A internacionalização no Adianti Framework é muito simples, e controlada por meio da classe `ApplicationTranslator`.

A classe `ApplicationTranslator`, que fica na pasta `app/lib`, e, portanto, dentro do espaço da aplicação, contém um conjunto de vetores indexados pelo idioma. No exemplo a seguir, temos três mensagens em inglês e três em português. As mensagens devem ser cadastradas na mesma ordem nos dois idiomas. Para adicionarmos novos idiomas, basta adicionarmos novos elementos no vetor, com um índice diferente. Esta classe acompanha uma série de métodos que a tornam funcional, como `translate()` e `translateTemplate()`. Ao final, ela acompanha a função `_t()`, função a ser chamada nas classes da aplicação para realizar a tradução de um termo.

`app/lib/util/ApplicationTranslator.class.php`

```
<?php
class ApplicationTranslator
{
    private static $instance;
    private $lang;

    private function __construct()
    {
        $this->messages = [];
        $this->messages['en'] = [];
        $this->messages['en'][1] = 'File not found';
        $this->messages['en'][2] = 'Search';
        $this->messages['en'][3] = 'Register';

        $this->messages['pt'] = [];
        $this->messages['pt'][1] = 'Arquivo não encontrado';
        $this->messages['pt'][2] = 'Buscar';
        $this->messages['pt'][3] = 'Cadastrar';

        $this->messages['es'] = [];
        $this->messages['es'][1] = 'Archivo no encontrado';
        $this->messages['es'][2] = 'Buscar';
        $this->messages['es'][3] = 'Registrar';
    }

    public static function getInstance()
    public static function setLanguage($lang)
    public static function getLanguage()
    static public function translate($word, $param1 = NULL, $param2 = NULL,
                                    $param3 = NULL)
    static public function translateTemplate($template)
}

function _t($msg, $param1 = null, $param2 = null, $param3 = null)
{
    return ApplicationTranslator::translate($msg, $param1, $param2, $param3);
}
```

Já vimos como definir a nossa classe de internacionalização, incluindo os termos traduzidos, agora vamos ver como utilizá-los na prática. Em primeiro lugar, precisamos definir qual será o idioma utilizado pela nossa aplicação. Para alterar esta definição, basta editar o arquivo `application.ini` e alterar a variável `language`.

app/config/application.ini

```
timezone = America/Sao_Paulo
language = pt
application = library
theme = theme3
```

Ao escrevermos um controlador de páginas, seguidamente utilizamos termos que são exibidos para o usuário, principalmente em rótulos de texto (`TLabel`) e títulos de datagrids (`TGridColumn`). Nestes casos, os termos podem ser exibidos para o usuário conforme o idioma definido na configuração.

Para que um termo seja traduzido pelo Adianti Framework, em primeiro lugar ele deve ser escrito em inglês. A partir do inglês, as traduções são realizadas para os outros idiomas, conforme a seleção de idioma. A tradução é realizada pela função `_t()`, que integra a classe `ApplicationTranslator`, vista anteriormente. A função `_t()` recebe um termo em inglês, e localiza a sua posição (índice) no vetor de termos em inglês (`$this->messages['en']`). A partir deste índice, ela localiza o termo traduzido no vetor em outro idioma, por exemplo, o português (`$this->messages['pt']`). Então, o termo é exibido para o usuário final, já traduzido. Na classe de exemplo a seguir, os termos em destaque utilizam a função de tradução, sendo traduzidos.

Exemplo de tradução

```
<?php
class Exemplo extends TPage
{
    protected $form;
    function __construct()
    {
        parent::__construct();
        ...
        $table->addRowSet(new TLabel( _t('Login') . ': '), $user);
        $table->addRowSet(new TLabel( _t('Password') . ': '), $pass);
        $table->addRowSet(new TLabel( _t('Language') . ': '), $lang);
```

Utilizar uma função de tradução dentro das classes controladoras é relativamente simples, uma vez que estamos dentro do universo PHP. Mas os menus e templates da aplicação, que estão escritos em HTML, também são responsáveis por apresentar termos aos usuários. Nestes casos, basta utilizarmos a notação indicada em negrito, a seguir, `_t{palavra}`, para obter a palavra correspondente, traduzida pela aplicação.

Obs: Ao utilizar a classe `THtmlRenderer`, a tradução precisa ser habilitada pelo método `enableTranslation()`. Já a tradução do template principal da aplicação é feita pelo método `translateTemplate()`, executado na página de entrada `index.php`.

menu.xml

```
<menu>
<menuitem label='_t{Cataloging}'>
    <icon>fa:book fa-fw</icon>
    <menu>
        <menuitem label='_t{Books}'>
            <icon>fa:th fa-fw</icon>
            <action>BookList</action>
        </menuitem>
        <menuitem label='_t{Collections}'>
            <icon>fa:th fa-fw</icon>
            <action>CollectionFormList</action>
        </menuitem>
        <menuitem label='_t{Classifications}'>
            <icon>fa:th fa-fw</icon>
            <action>ClassificationFormList</action>
        </menuitem>
        <menuitem label='_t{Subjects}'>
            <icon>fa:th fa-fw</icon>
            <action>SubjectList</action>
        </menuitem>
    </menu>
</menuitem>
</menu>
```

6.2.7 Multi idioma

Ao trabalharmos com tradução, outra possibilidade que temos é de utilizar o multi idioma. Quando o multi idioma está ligado, é solicitado que o usuário escolha o idioma já na tela de login da aplicação. Este idioma selecionado é usado para traduções.

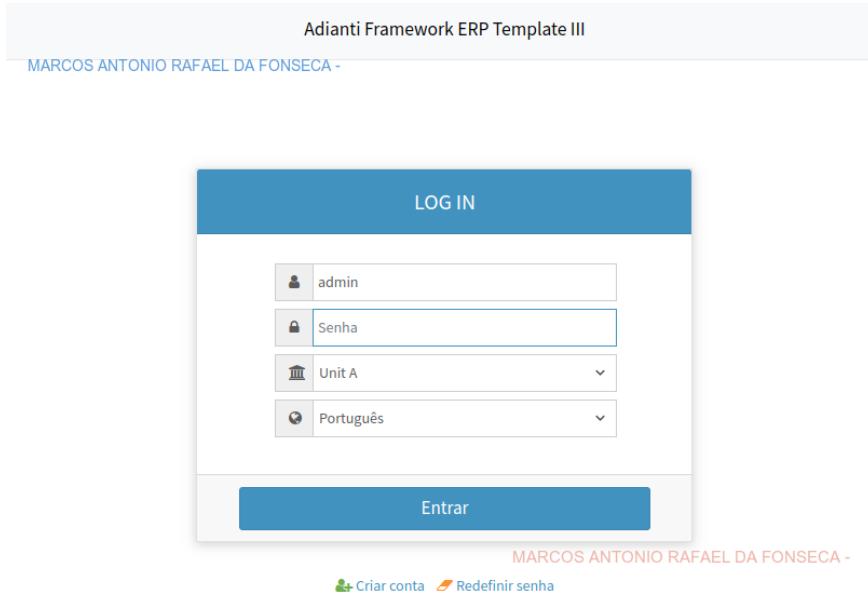


Figura 140 Seleção de idioma na tela de login

Para ligar o multi idioma, basta ligar a variável `multi_lang` no `application.ini`. As opções disponíveis para o usuário devem estar contidas na variável `lang_options`.

app/config/application.ini

```
[general]
multi_lang = "1"
lang_options[pt] = Português
lang_options[en] = English
lang_options[es] = Español
```

O Framework fará a seleção do idioma, baseado no que o usuário selecionou na tela de login. Este controle é realizado no `engine.php`. Você não precisa realizar nenhuma ação quanto à isto. Colocamos o trecho a seguir, apenas para esclarecer o funcionamento.

engine.php

```
ApplicationTranslator::setLanguage( TSession::getValue('user_language'), true );
```

6.2.8 Definindo uma autenticação alternativa (LDAP)

O Framework sai de fábrica com seu método nativo de autenticação de usuário e senha. Mas é possível indicar um método alternativo de autenticação, tal como AD/LDAP, facilmente. Nesta sessão, vamos instrui-lo a criar um serviço alternativo de autenticação baseado em LDAP. Para tal, precisaremos indicar ao Framework a utilização de uma classe alternativa de serviço de autenticação.

Para utilizar uma classe alternativa de serviço de autenticação, basta informarmos a classe de autenticação no arquivo `application.ini`, na seção `[permission]`. A propriedade `auth_service` deve conter o nome de uma classe de serviço que fará a autenticação. Esta classe deve possuir um método de autenticação (`authenticate`), que fará a autenticação de usuário e senha. Em seguida, esta classe será explicada.

app/config/application.ini

```
[permission]
auth_service = LdapAuthenticationService
```

Neste exemplo, estamos implementando um controle de login por AD/LDAP, cuja configuração estará armazenada no arquivo `app/config/ldap.ini`.

app/config/ldap.ini

```
server = ldap.company.com
port   = 389
domain = company.com
```

A classe que contém o serviço de autenticação deve possuir um método chamado `authenticate()`, que receberá o usuário e senha e deve proceder com a autenticação, lançando uma exceção caso as credenciais sejam inválidas, como no exemplo a seguir.

app/service/auth/LdapAuthenticationService.php

```
<?php
class LdapAuthenticationService
{
    public static function authenticate($user, $password)
    {
        $ldap = parse_ini_file('app/config/ldap.ini');
        $ds   = ldap_connect($ldap['server'], $ldap['port']);

        if ($ds)
        {
            if (@ldap_bind($ds, $user . '@' . $ldap['domain'], $password))
            {
                return true;
            }
        }
        throw new Exception(_t('Invalid LDAP credentials'));
    }
}
```

Obs: Esta implementação para autenticação AD/LDAP é só uma sugestão. Cada servidor AD/LDAP possui especificidades que devem ser observadas no momento da autenticação, fazendo que esta classe precise ser adaptada conforme o caso.

6.2.9 O index

O `index.php` é o arquivo para carregamento da página inicial da aplicação. Quando estudamos o arquivo `index.php` pela primeira vez, estávamos analisando o Framework puro, ou seja, sem os controles de permissão de acesso. Naquele caso, o `index.php` carregava qualquer conteúdo (classe) indiscriminadamente. O `index.php` que acompanha o Template, possui uma série de controles. Conforme o usuário está ou não logado, ele carrega diferentes arquivos de layout, como veremos a seguir.

O `index` começa pela inicialização do Framework, que se dá por meio da requisição ao arquivo `init.php`. Em seguida, algumas variáveis são iniciadas como: `$theme` (tema definido no `application.ini`), `$class` (a classe identificada na URL), e `$public` (verifica se a classe solicitada é pública).

Em seguida, uma sessão é iniciada, e é verificado se o usuário está logado, o que é feito pelo método `TSession::getValue('logged')`. Caso esteja logado, é carregado o layout padrão (`layout.html`). Caso contrário, uma outra verificação é realizada, a seguir.

Caso o usuário não esteja logado, existem duas possibilidades de caminho. A primeira é termos um layout público habilitado (`public_view`) no `application.ini`. Em seções seguintes, abordaremos o que é um layout ou visão pública. Basicamente se trata de uma página para acesso à páginas públicas, ou seja, um layout não autenticado. Caso o layout público esteja habilitado, então ele é carregado (`public.html`), caso contrário, então é carregado o layout de login (`login.html`).

Caso o usuário esteja logado, é carregado o menu hierárquico por meio da classe `AdiantiMenuBuilder`, a partir do arquivo `menu.xml`. Esta classe lê o arquivo `menu.xml`, filtra-o, e gera o menu somente com as opções que o usuário tem acesso. Para tal, internamente a classe utiliza um método chamado `SystemPermission::checkPermission()`, que é responsável por verificar se o usuário tem acesso a um determinado programa (classe de controle) ou não. Assim, o menu é montado somente com as opções que o usuário tem acesso. O menu é injetado no layout por meio da substituição da variável `{MENU}`. O mesmo procedimento ocorre se o layout público estiver habilitado. Neste caso, são lidos outros arquivos para layout (`public.html`), e menu (`menu-public.xml`).

O método `ApplicationTranslator::translateTemplate()` realiza tradução de termos do template quando necessário. Para tal, os termos devem estar entre `_t{termo}`. Já o método `AdiantiTemplateParser::parse()` realiza substituição de uma série de variáveis no Template, que dependem do perfil do usuário, tais como `{login}`, `{username}`, `{usermail}`, e outras globais como `{LIBRARIES}`, `{class}`, e `{template}`.

Após realizar uma série de substituições, ao final do script é feito o carregamento da classe requisitada pela URL. Caso o usuário esteja logado ou a classe requisitada é de acesso público (`$public`), então a classe é carregada por meio do método `loadPage()`. Caso o usuário não esteja logado, existem ainda duas possibilidades. Se a visão pública estiver ligada, o usuário é direcionado à uma página de entrada configurável no `application.ini`, na variável `public_entry`. Se a visão pública estiver desligada (caso mais comum), a classe de login (`LoginForm`) é carregada.

index.php

```
<?php
require_once 'init.php';
$theme = $ini['general']['theme'];
$class = isset($_REQUEST['class']) ? $_REQUEST['class'] : '';
$public = in_array($class, $ini['permission']['public_classes']);
new TSession;
ApplicationTranslator::setLanguage( TSession::getValue('user_language'), true );

if ( TSession::getValue('logged') ) { // usuário logado
    $content = file_get_contents("app/templates/{$theme}/layout.html");
    $menu = AdiantiMenuBuilder::parse('menu.xml', $theme);
    $content = str_replace('{MENU}', $menu, $content);
}
else {
    // a visão pública está ligada
    if (isset($ini['general']['public_view']) && $ini['general']['public_view'] == '1')
    {
        $content = file_get_contents("app/templates/{$theme}/public.html");
        $menu = AdiantiMenuBuilder::parse('menu-public.xml', $theme);
        $content = str_replace('{MENU}', $menu, $content);
    }
    else // visão pública desligada → vai para login
    {
        $content = file_get_contents("app/templates/{$theme}/login.html");
    }
}
```

```

// traduz template
$content = ApplicationTranslator::translateTemplate($content);

// realiza substituições no template
$content = AdiantiTemplateParser::parse($content);

// saída para template
echo $content;

// se tem permissão
if (TSession::getValue('logged') OR $public)
{
    if ($class)
    {
        $method = isset($_REQUEST['method']) ? $_REQUEST['method'] : NULL;
        AdiantiCoreApplication::loadPage($class, $method, $_REQUEST);
    }
}
else
{
    // se visão pública está ligada
    if (isset($ini['general']['public_view']) && $ini['general']['public_view'] == '1')
    {
        if (!empty($ini['general']['public_entry']))
        {
            AdiantiCoreApplication::loadPage($ini['general']['public_entry'], '', $_REQUEST);
        }
    }
    else
    {
        // vai para formulário de login
        AdiantiCoreApplication::loadPage('LoginForm', '', $_REQUEST);
    }
}

```

Uma das verificações realizadas pelo `index.php` é se a classe requisitada tem acesso público, o que é controlado pela variável `$ini['permission']['public_classes']`. Esta variável é resultado da leitura do arquivo `application.ini`.

app/config/application.ini

```

[permission]
public_classes[] = PublicForm
public_classes[] = PublicView

```

6.2.10 O engine

O engine (`engine.php`) é o motor de execução do Framework, por onde todas as requisições de páginas passam após a página inicial estar carregada. É nele que deve ser implementado o controle de acesso. Chamadas de métodos como `AdiantiCoreApplication::loadPage()` também passam pelo `engine.php`. Se você monitorar o fluxo de rede (network) durante a execução de um programa, verá ele sendo solicitado várias vezes.

No engine é verificado se o usuário está logado. Se ele está logado (`logged`), então é verificado se o mesmo possui acesso ao programa solicitado, o que é realizado observando a variável `$programs`, que é composta pelos programas que o usuário possui acesso (`TSession::getValue('programs')`), que dependem do cadastro do

usuário e dos grupos de acesso, e também das permissões default (`getDefaultValuePermissions()`) que são alguns programas que todos usuários logados possuem acesso. Caso o programa a ser acessado está dentro desta lista, ou é um programa público (`$public`), que por sua vez é definido no `application.ini`, o acesso é liberado e ele é executado (`parent::run()`).

É importante lembrar que a variável de sessão `programs` é gravada pelo formulário de login. Caso ele tenha acesso ou o programa é público, o programa é executado (`parent::run()`), caso contrário uma mensagem é apresentada (*Permission denied*). Caso o usuário não esteja logado, mas o programa solicitado é o formulário de login (`LoginForm`), sua execução é permitida. Caso a execução caia no último `ELSE`, então uma mensagem de erro é apresentada (*Permission denied*).

engine.php

```
<?php
require_once 'init.php'; // inicialização do Framework

class TApplication extends AdiantiCoreApplication
{
    public static function run($debug = null)
    {
        new TSession;

        // aplica o idioma selecionado pelo usuário, se aplicável (multi-idioma)
        ApplicationTranslator::setLanguage( TSession::getValue('user_language'), true );

        if ($_REQUEST) {
            $ini = AdantiApplicationConfig::get();

            $class = isset($_REQUEST['class']) ? $_REQUEST['class'] : '';
            $public = in_array($class, $ini['permission']['public_classes']);
            $debug = is_null($debug)? $ini['general']['debug'] : $debug;
            if (TSession::getValue('logged')) // logged
            {
                $programs = (array) TSession::getValue('programs');
                $programs = array_merge($programs, self::getDefaultPermissions());

                if( isset($programs[$class]) OR $public )
                {
                    parent::run($debug);
                }
                else
                {
                    new TMessage('error', _t('Permission denied'));
                }
            }
            else if ($class == 'LoginForm' OR $public )
            {
                parent::run($debug);
            }
            else {
                new TMessage('error', _t('Permission denied'), new
TAction(array('LoginForm','onLogout')));
            }
        }
        // ...
    }

    TApplication::run(TRUE);
}
```

6.2.11 Visão pública

O Template sai de fábrica com apenas duas visualizações: a Tela de login, e a interface completa, onde após logado, você tem acesso aos menus e as páginas conforme seu nível de permissão. Porém, você pode habilitar um terceiro tipo de visualização que é a Visão pública, ou layout público. Esta visão oferece a possibilidade de acesso ao usuário final à programas públicos por meio de um menu público, antes do login no sistema.

Se a visão pública estiver ligada, então a visualização inicial do usuário já será do sistema com um menu à esquerda. Neste menu, serão listados os programas de acesso público. No canto superior direito da tela, existirá um botão para o usuário efetuar login no sistema. Quando o usuário clicar para efetuar login, o formulário de login será carregado dentro desta interface, e após o login, o usuário terá acesso ao sistema completo.

Para habilitar a visão pública, é necessário ligar a variável `public_view` no `application.ini`. Também é possível definirmos um programa público inicial para o usuário, ou seja, um programa default na abertura do sistema, por meio da variável `public_entry`. O menu público do sistema será lido a partir do arquivo `menu-public.xml`. Este menu conterá as opções públicas. Para liberar a permissão pública para um programa, basta acrescentá-lo na lista de `public_classes` do `application.ini`. A visão pública utiliza o arquivo de layout `public.html`, e não o `layout.html` para exibir o conteúdo em tela. Assim, qualquer personalização visual da visão pública, deve ser realizada neste arquivo.

app/config/application.ini

```
[permission]
public_view = "1"
public_entry = "PublicView"

[permission]
public_classes[] = PublicForm
public_classes[] = PublicView
```

A seguir, um exemplo de visão pública.

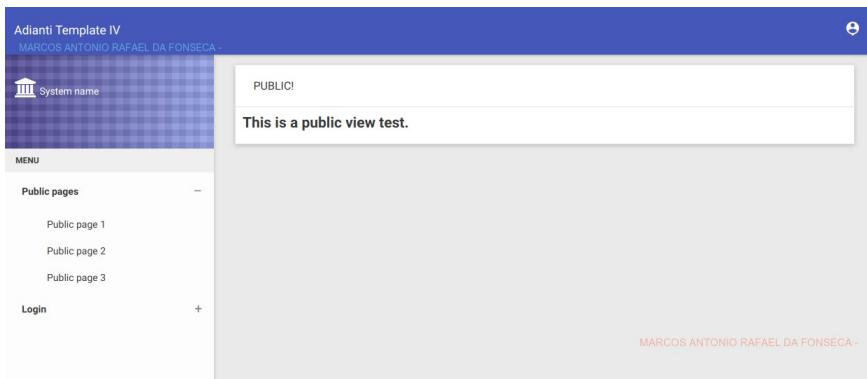


Figura 141 Visão pública

6.2.12 Cadastro de programas

O Template já acompanha uma série de cadastros administrativos que permitem definir as permissões da aplicação. Para acessá-los, basta realizar login no sistema, com o usuário **admin**, cuja senha já foi apontada anteriormente. O primeiro cadastro é o de programas. Qualquer classe para poder ser acessada pelo usuário, deve primeiramente ser cadastrada como um programa. A seguir, temos a lista de programas.

MARCOS ANTONIO RAFAEL DA FONSECA -					
	ID	Classe de controle	Nome	Caminho menu	
	1	SystemGroupForm	System Group Form		
	2	SystemGroupList	System Group List	Administração » Grupos	
	3	SystemProgramForm	System Program Form		
	4	SystemProgramList	System Program List	Administração » Programas	
	5	SystemUserForm	System User Form		
	6	SystemUserList	System User List	Administração » Usuários	
	7	CommonPage	Common Page	Common pages » Common page 3	
	8	SystemPHPInfoView	System PHP Info	Administração » PHP Info	
	9	SystemChangeLogView	System ChangeLog View	Logs » Log de alterações	
	10	WelcomeView	Welcome View		

1 a 10 de 40 registros MARCOS ANTONIO RAFAEL DA FONSECA -

1 2 3 4 5 6 7 8 9 10

Figura 142 Lista de programas

No cadastro de programas, devemos registrar cada classe (página) criada, dando um nome, e definindo a sua classe de controle. Você já pode neste momento, conceder a permissão para um ou mais grupos de usuários.

Programa	
MARCOS ANTONIO RAFAEL DA FONSECA -	
ID	<input type="text" value="1"/>
Classe de controle	<input type="text" value="SystemGroupForm"/> x ▼
Nome	<input type="text" value="System Group Form"/>
Grupos	
<input checked="" type="checkbox"/> Admin	<input type="checkbox"/> Standard
MARCOS ANTONIO RAFAEL DA FONSECA -	
<input type="button" value="Salvar"/> <input type="button" value="Limpar"/> <input type="button" value="Voltar"/>	

Figura 143 Cadastro de programas

6.2.13 Cadastro de grupos

Outro cadastro administrativo é o de grupos de usuário. Um grupo de usuários define um conjunto de permissões. Um usuário pode fazer parte de um ou mais grupos, herdando suas permissões. Na figura a seguir, você pode conferir a listagem de grupos.

MARCOS ANTONIO RAFAEL DA FONSECA -		<input type="button" value="Exportar"/>
ID	Nome	
	1 Admin	
	2 Standard	

1 a 2 de 2 registros [MARCOS ANTONIO RAFAEL DA FONSECA -](#)

[1](#) [2](#) [3](#) [4](#) [5](#) [6](#) [7](#) [8](#) [9](#) [10](#)

Figura 144 Lista de grupos

No cadastro de grupos, que pode ser visto na figura a seguir, podemos relacionar uma série de programas. É importante lembrar que um grupo pode conter muitos programas, o que se dá pelo relacionamento de agregação entre `SystemGroup` e `SystemProgram`.

Grupo		
MARCOS ANTONIO RAFAEL DA FONSECA -		
ID	1	
Nome	Admin	

[Programas](#) [Usuários](#)

ID	Nome	Buscar	Caminho menu
<input checked="" type="checkbox"/> 1	System Group Form		
<input type="checkbox"/> 2	System Group List		Administração » Grupos
<input checked="" type="checkbox"/> 3	System Program Form		
<input type="checkbox"/> 4	System Program List		Administração » Programas
<input checked="" type="checkbox"/> 5	System User Form		
<input type="checkbox"/> 6	System User List		Administração » Usuários
<input type="checkbox"/> 7	Common Page		Common pages » Common page 3
<input type="checkbox"/> 8	System PHP Info		Administração » PHP Info
<input type="checkbox"/> 9	Custom Channel on View		Lore - Lore da alteração

MARCOS ANTONIO RAFAEL DA FONSECA -

[Salvar](#) [Limpar](#) [Voltar](#)

Figura 145 Cadastro de grupos

6.2.14 Cadastro de usuários

O cadastro de usuários permite relacionar um usuário a um ou vários grupos, a um ou vários programas, e a uma ou várias unidades. O usuário terá automaticamente acesso à todos os programas dos grupos que ele fizer parte. Podemos marcar vários grupos nas checkboxes, o que é representado pela agregação entre `SystemUser` e `SystemGroup`.

O usuário poderá estar vinculado à várias unidades. As unidades selecionadas representam aquelas que ele poderá selecionar na tela de login, quando o multi unidade estiver ligado. Esta relação é representada pela agregação entre `SystemUser` e `SystemUnit`.

Também podemos adicionar vários programas individuais ao usuário na área de programas, o que é representada pela agregação entre `SystemUser` e `SystemProgram`. Assim, além dos programas do grupo, ele terá acesso à programas individuais.

É importante lembrar que, ao vincularmos o usuário a um grupo, este terá acesso a todos os programas que fazem parte daquele grupo.

The screenshot shows a user creation form with the following fields:

- Usuário**: A header section containing the user's name: MARCOS ANTONIO RAFAEL DA FONSECA -.
- ID**: Value 1.
- Nome**: Value Administrator.
- Login**: Value admin.
- Email**: Value admin@admin.net.
- Unidade principal**: A dropdown menu currently empty.
- Tela inicial**: Value Welcome View.
- Senha**: An empty password field.
- Confirma senha**: An empty confirmation password field.
- Unidades**: A section with checkboxes for Unit A (checked) and Unit B.
- Grupos**: A section with checkboxes for Admin (checked) and Standard.
- Programas**: A section showing a list of programs:

ID	Nome	Caminho menu
1	System Group Form	
2	System Group List	Administração > Grupos
3	System Program Form	
4	System Program List	Administração > Programas
- Buttons**: Save, Clear, and Back.

Figura 146 Cadastro de usuários

Ainda referente ao gerenciamento de usuários, na listagem de usuários várias operações podem ser realizadas, tais como: a exportação da lista de usuários em PDF, CSV, e XML; a busca com filtro de usuários; a edição do usuário; sua exclusão; podemos ativar e desativar um usuário, e também podemos personificá-lo, ou seja, acessar com seu login.

The screenshot shows the 'User' management interface. At the top, there is a search form with fields for 'Id', 'Nome', 'Email', and 'Ativo' (Active status). Below the search form are two buttons: 'Buscar' (Search) and 'Novo' (New). The main area displays a grid of users with columns: Id, Nome, Login, Email, and Ativo. Each row has edit and delete icons. The first user listed is 'Administrator' with Id 1, and the second is 'User' with Id 2. Both users have 'admin' as their login and 'Sim' as their active status. At the bottom of the grid, it says '1 a 2 de 2 registros' (1 to 2 of 2 records) and 'MARCOS ANTONIO RAFAEL DA FONSECA -'. Below the grid is a navigation bar with pages 1 through 10, where page 1 is highlighted.

Figura 147 Lista de usuários

6.2.15 Cadastro de unidades

O cadastro de unidades permite criar unidades organizacionais, que podem representar filiais ou setores de uma empresa. Usuários podem ser relacionados a uma ou várias unidades. Quando o multi unidade estiver ligado, o usuário selecionará a unidade na tela de login, e a informação da unidade escolhida fica disponível na variável de sessão `userunitid`, e pode ser utilizada para filtros de datagrids, por exemplo.

The screenshot shows the 'Unidade' (Unit) registration interface. At the top, it says 'Unidade' and 'MARCOS ANTONIO RAFAEL DA FONSECA -'. The form has fields for 'Id' (set to 1) and 'Nome' (set to 'Departamento de RH'). At the bottom are three buttons: 'Salvar' (Save), 'Limpar' (Clear), and 'Voltar' (Back). A message 'MARCOS ANTONIO RAFAEL DA FONSECA -' is also visible at the bottom right.

Figura 148 Cadastro de unidades

6.2.16 Database explorer

O Template conta também com um navegador de base de dados (Database Explorer). Por meio dele, você pode selecionar uma conexão de banco de dados. A partir desta seleção, ele listará todas as tabelas daquela conexão. Após selecionar a tabela, os dados serão exibidos no centro da tela com uma paginação.

Você pode ainda clicar sobre um conector de banco de dados ou de uma tabela, e exportá-los em CSV ou SQL. Na tabela do centro da tela, é possível filtrar os dados, paginá-los, e ainda clicar no botão SQL, que carregará um console SQL para realizar consultas personalizadas sobre aquela tabela.

The screenshot shows the Database Explorer interface with two main sections: 'Banco de dados' (left) and a detailed view of the 'permission > system_program' table (right).

Banco de dados:

- Connections listed: MARCOS ANTONIO RAFAEL DA FONSECA (selected), communication (sqlite), log (sqlite), permission (sqlite) (highlighted with an orange border), unit_a (sqlite), unit_b (sqlite).
- Tables listed: system_group, system_program (highlighted with an orange border), system_unit, system_preference, system_user_group.

(permission > system_program) View:

id	name	controller
1	System Group Form	SystemGroupForm
2	System Group List	SystemGroupList
3	System Program Form	SystemProgramForm
4	System Program List	SystemProgramList
5	System User Form	SystemUserForm
6	System User List	SystemUserList
7	Common Page	CommonPage
8	System PHP Info	SystemPHPInfoView
9	System ChangeLog View	SystemChangeLogView
10	Welcome View	WelcomeView

Pagination at the bottom: MARCOS ANTONIO RAFAEL DA FONSECA - 1 2 3 4 5 6 7 8 9 10

Figura 149 Database explorer

6.2.17 Painel de SQL

O painel de SQL é um recurso que permite ao administrador realizar consultas rápidas nas tabelas da base de dados. Basta selecionar o conector e a tabela, que a ferramenta gera um SQL básico que você pode alterar para acrescentar seus próprios filtros.

id	name
1	Admin
2	Standard

2 registros exibidos

Figura 150 Painel de SQL

6.2.18 Preferências

O formulário de preferências do sistema permite definir preferências gerais. No momento, ele é distribuído com algumas preferências de envio de e-mail, mas o desenvolvedor pode facilmente adicionar novos campos a esse formulário.

O formulário de preferências grava as informações na tabela `system_preference` que possui apenas dois atributos: `id` e `value`. Assim, caso você precise de novas preferências, não é necessário criar tabelas ou colunas para armazenar estas informações, bastando criar novos campos no formulário, que serão armazenados na forma de novos registros. Para obter uma preferência de qualquer ponto da aplicação basta executar o método `getPreference()` da classe `SystemPreference`, identificando a chave da preferência.

Obter preferência

```
SystemPreference::getPreference('smtp_user');
```

Na figura a seguir, temos a tela de configuração de Preferências do Sistema.

The screenshot shows a 'Preferências' (Preferences) window. At the top, it displays the name 'MARCOS ANTONIO RAFAEL DA FONSECA'. Below this, there are several input fields for SMTP settings:

- E-mail de origem: [empty input field]
- Autentica SMTP: A dropdown menu currently set to 'ssl://smtp.gmail.com, tls://server.company.com'
- Host SMTP: Input field containing 'ssl://smtp.gmail.com, tls://server.company.com'
- Porta SMTP: [empty input field]
- Usuário SMTP: [empty input field]
- Senha SMTP: [empty input field]
- Email de suporte: [empty input field]

At the bottom right of the form area, the name 'MARCOS ANTONIO RAFAEL DA FONSECA' is repeated. A blue 'Salvar' (Save) button is located at the bottom left.

Figura 151 Preferências do sistema

6.2.19 Editor de menu

O editor de menu permite ao administrador adicionar novos módulos ou opções ao menu do sistema. Para que isto seja possível, é importante que o programa (classe de controle) exista, esteja registrada no cadastro de programas, e tenha liberação para pelo menos algum usuário ou grupo. Caso contrário, o ato de acrescentar no menu não terá efeito algum, pois ninguém terá permissão de acessar aquele programa.

O editor de menu armazena as definições de tela no arquivo `menu.xml`. Você pode editar este arquivo manualmente caso necessário, o que o editor de menu oferece é uma forma cômoda para criar novos itens. É imprescindível que o arquivo `menu.xml` tenha permissão de escrita para que o editor de menu funcione.

Ordenação	Rótulo	Ação	Ícone	Cor
+ _t[Administration]	_t[Administration]		<i>fa-univers... x</i>	
+ Dashboard	Dashboard	SystemAdministrati...	<i>fa-file-cod... x</i>	
+ _t[Programs]	_t[Programs]	SystemProgramList	<i>fa-file-cod... x</i>	
+ _t[Groups]	_t[Groups]	SystemGroupList	<i>fa-users x</i>	
+ _t[Units]	_t[Units]	SystemUnitList	<i>fa-univers... x</i>	
+ _t[Users]	_t[Users]	SystemUserList	<i>fa-user x</i>	
+ _t[Database explorer]	_t[Database explorer]	SystemDatabaseExp...	<i>fa-database x</i>	
+ _t[SQL Panel]	_t[SQL Panel]	SystemSQLPanel	<i>fa-table x</i>	
+ _t[Menu editor]	_t[Menu editor]	SystemMenuEditor	<i>fa-pencil... x</i>	
+ PHP Info	PHP Info	SystemPHPInfoView	<i>fa-info-circle x</i>	
+ _t[Preferences]	_t[Preferences]	SystemPreferenceC...	<i>fa-cog x</i>	
+ _t[System information]	_t[System information]	SystemInformation	<i>fa-cog x</i>	MARCOS ANTONIO RAFAEL DA FONSECA -
+ _t[Documents]	_t[Documents]		<i>fa-file-text-o x</i>	

Figura 152 Editor de menu

6.3 Configuração e depuração

Nesta seção vamos verificar algumas ferramentas do Template que auxiliam o desenvolvedor a verificar como está a configuração do ambiente e também a depurar a execução do programa.

6.3.1 PHP Modules

A opção PHP Modules, que consta no rodapé do sistema, apresenta um resumo sobre as configurações recomendadas para desenvolvimento e produção, bem como uma lista resumida de quais módulos do PHP estão instalados.

The screenshot shows the AdiantiTemplate III application interface. On the left, there is a sidebar with navigation links such as Dashboard, Programas, Grupos, Unidades, Usuários, Database explorer, Painel SQL, Editor de menu, PHP Info, Preferências, and Informações do sistema. The main content area has two tabs: 'PHP Directives' and 'PHP Modules'. The 'PHP Directives' tab displays a table with columns for Directive, Current, Development, and Production. The 'PHP Modules' tab lists various PHP modules with their status (enabled or disabled).

DIRECTIVE	CURRENT	DEVELOPMENT	PRODUCTION
error_reporting	E_ALL	E_ALL	E_ALL & ~E_DEPRECATED & ~E_STRICT
display_errors	On	On	Off
log_errors	On	On	On
output_buffering	4096	4096	4096
opcache.enable	Off	On	On

A message bar at the bottom says: "A localização atual do php.ini é /usr/local/php7/lib/php.ini". Another message bar below it says: "Note: error_reporting and display_errors are automatic enabled when debug=1 in application.ini".

GENERAL		DATABASE	
✓ MBString (mbstring)		✓ PDO	
✓ CURL (curl)		✓ PDO SQLite (pdo_sqlite)	
✓ DOM (dom)		✓ PDO MySql (pdo_mysql)	
✓ XML (xml)		✓ PDO PostgreSQL (pdo_pgsql)	
✓ ZIP (zip)		✓ PDO Oracle (pdo_oci)	
✓ JSON (json)		✓ PDO Sql Server via dblib (pdo_dblib)	
✓ LibXML (libxml)		✓ PDO Sql Server via sqlsrv (pdo_sqlsrv)	
✓ OpenSSL (openssl)		✗ PDO Firebird (firebird)	
✓ SimpleXML (SimpleXML)		✓ PDO ODBC (odbc)	MARCOS ANTONIO RAFAEL DA FONSECA -
✓ FileInfo (fileinfo)			

Figura 153 PHP Modules

6.3.2 PHP Info

A opção PHPInfo no menu Administração exibe toda a configuração de instalação do PHP, suas diretivas, e módulos instalados de maneira detalhada. Utilize esta funcionalidade para verificar se algum módulo está faltando em sua instalação, e qual a localização do `php.ini`, que é o local onde as configurações são realizadas, dentre outros.

A localização atual do php.ini é /usr/local/php7/lib/php.ini

PHP Version 7.2.16

System	Linux adianti 4.15.0-65-generic #74-Ubuntu SMP Tue Sep 17 17:06:04 UTC 2019 x86_64
Build Date	Mar 23 2019 19:20:32
Configure Command	'./configure' '--enable-cgi' '--enable-cli' '--prefix=/usr/local/php7' '--with-apxs2=/usr/bin/apxs2' '--with-mysqli=mysqlnd' '--with-pdo-mysql=mysqlnd' '--with-pgsql' '--with-pdo-pgsql' '--with-pdo-sqlite' '--with-png-dir=/usr' '--with-jpeg-dir=/usr' '--with-zlib=/usr' '--with-bz2=/usr' '--with-libxml-dir=/usr' '--with-gd' '--with-freetype-dir=/usr/local/php7' '--enable-soap' '--enable-calendar' '--enable-sockets' '--enable-mbstring' '--enable-zip' '--enable-fpm' '--with-openssl' '--with-curl' '--with-unixODBC=/usr' '--with-pdo-odbc=unixODBC,/usr' '--with-ldap=shared,/usr'
Server API	Apache 2.0 Handler
Virtual Directory Support	disabled
Configuration File (php.ini) Path	/usr/local/php7/lib
Loaded Configuration File	/usr/local/php7/lib/php.ini
Scan this dir for additional .ini files	(none)
Additional .ini files parsed	(none)

Figura 154 PHP Info

6.3.3 Debug console

Outra opção cujo acesso é por meio do rodapé do sistema é o Debug console. Ao acionarmos o Debug console, um painel será aberto na parte inferior da tela, exibindo todos os Requests gerados pelo Framework, o endereço da URL, os parâmetros, e todo o body da requisição. É uma ferramenta simples de utilizar. Porém, recomendamos sempre que possível que o desenvolvedor use a aba Network do console do desenvolvedor do Chrome para depurar estas requisições, por se tratar de uma ferramenta mais completa.

ID	Nome	Caminho menu
1	Admin	
1	System Group Form	
2	System Group List	

```

Request URL
{
  class: "SystemGroupForm",
  method: "onSave"
}

Request Data
{
  id: "1",
  name: "Admin",
  search: "",
  check_program_list_1: "on",
  check_program_list_2: "on",
  check_program_list_3: "on",
  check_program_list_4: "on".
}
  
```

Figura 155 Debug console.

6.4 Módulo Logs

Nesta seção abordaremos o módulo de Logs, com sua modelagem de classes, tabelas, e funcionalidades.

6.4.1 Classes de Modelo

Em uma aplicação de negócios, é muito importante o registro de logs sobre operações, a fim de permitir uma melhor detecção de tentativas de acesso não autorizado, de tentativas de fraude, e até mesmo de falhas de desenvolvimento.

O Template permite o registro de logs de vários tipos. Cada tipo é gerenciado por uma classe de modelo. As classes e os tipos são:

- SystemAccessLog**: gerencia logs de acesso, login e logout dos usuários no sistema, bem como os momentos dos acessos;
- SystemSqlLog**: gerencia logs de SQL e permite registrar os comandos SQL (**INSERT**, **UPDATE**, **DELETE**), gerados internamente pelo framework, bem como o usuário logado, e o momento do comando;
- SystemChangeLog**: gerencia logs de mudanças e permite registrar mudanças (inserções, alterações, exclusões) em valores de campos da base de dados. Armazena os valores antigos e os novos de cada campo, bem como usuário, o momento da operação, dentre outras informações.
- SystemRequestLog**: gerencia logs de requisições HTTP, permite registrar o momento da requisição, o login de usuário, a sessão, endereços, parâmetros informados, cabeçalhos da requisição, dentre outros.

6.4.2 Classes de Serviço

Para registrar os logs, as classes de modelo não são chamadas diretamente. Para tal, internamente a responsabilidade pelo registro recai sobre classes de serviço. As seguintes classes de serviço são utilizadas:

- SystemAccessLogService**: Classe de serviço que aciona a **SystemAccessLog** para registrar operações de login e logout, por meio de seus métodos **registerLogin()** e **registerLogout()**.
- SystemSqlLogService**: Classe de serviço que aciona a **SystemSqlLog** para registrar logs de SQL por meio de seu método **write()**.
- SystemChangeLogService**: Classe de serviço que aciona a **SystemChangeLog** para registrar logs de mudanças de registros por meio do método **register()**.
- SystemRequestLogService**: Classe de serviço que aciona a **SystemRequestLog** para registrar logs de requisições HTTP.

6.4.3 Modelo relacional

As classes de log manipulam dados que são registrados em tabelas do banco de log. O banco de log é configurado no arquivo `app/config/log.ini`. A instalação padrão do Template acompanha um banco de dados de log no formato SQLite, (`app/database/log.db`). Porém, neste mesmo diretório encontram-se as instruções SQL (`log.sql`) para criação do banco de logs em outros sistemas de bancos de dados. Quando utilizar outro banco de dados, você precisará alterar a configuração de acesso ao banco de logs. A seguir, o arquivo de configuração de acesso ao banco de logs.

Obs: Não exclua sob hipótese alguma o arquivo `log.ini`, uma vez que existem vários pontos dentro do template que referenciam este arquivo no momento de abrir conexão com a base de dados de log.

app/config/log.ini

```
host    =
port   =
name   = app/database/log.db
user   =
pass   =
type   = sqlite
```

A seguir, estão as tabelas de registro de logs. Basicamente, temos as tabelas:

- system_access_log**: armazena os acessos dos usuários no sistema. Registra informações como o id da sessão, o login, os momentos de login e de logout, ip;
- system_sql_log**: armazena os comandos SQL (`INSERT`, `UPDATE`, `DELETE`) gerados pelo framework. Registra informações como o momento do log, o login do usuário, o nome da conexão com o banco, o comando SQL, a sessão, o trace do log, e o tipo de comando (`INSERT`, `UPDATE`, `DELETE`);
- system_change_log**: armazena os logs de mudanças (inserções, alterações, exclusões) em valores de campos da base de dados. Registra informações como o momento do log, o login do usuário, o nome da tabela, o campo de chave primária, o valor da chave primária, a operação realizada (inserção, alteração, exclusão), o nome da coluna alterada, o valor antigo, bem como o valor novo, a sessão, etc.
- system_request_log**: armazena os logs de requisições. Toda requisição HTTP, seja pela navegação convencional, ou por chamadas de Web Services, mas também chamadas da API pela linha de comando são registradas no log. Informações como o momento da requisição, o login, a sessão de uso, IP, host, servidor, porta, endereço da requisição, parâmetros, dentre outros são registrados.

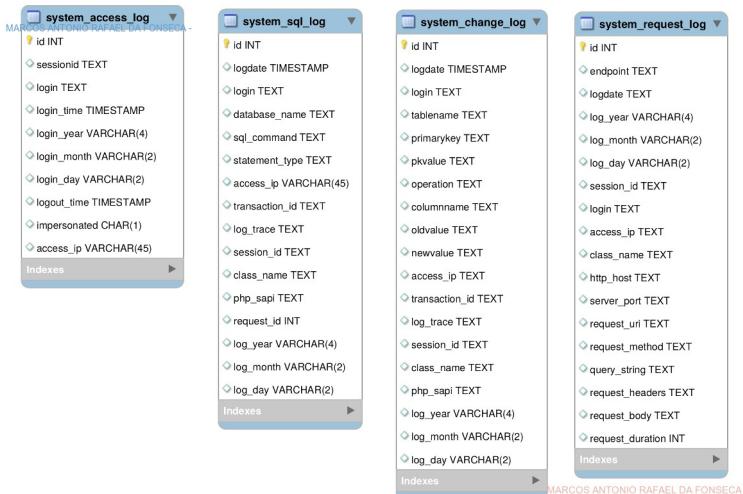


Figura 156 Tabelas de registro de log

6.4.4 Logs de acesso

Você não precisa habilitar os logs de acesso, pois os mesmos já estão habilitados por padrão na classe `LoginForm`. Lá, você encontra as chamadas para os métodos `SystemAccessLogService::registerLogin()` para registrar o login, bem como `SystemAccessLogService::registerLogout()` para registrar o logout do usuário. Estes métodos registram os acessos dos usuários automaticamente na tabela `system_access_log`. Caso você queira registrar de maneira diferente os acessos, pode estudar a classe `SystemAccessLog`, e `SystemAccessLogService` para criar outra classe de log de acesso.

O Template já acompanha uma série de telas para monitorarmos os logs gerados. Estas telas podem ser acessadas por meio do menu “Logs”. A tela de consulta ao log de acessos ao sistema contém informações como o id da sessão, o login do usuário, o momento do login e o momento do logout. Pode-se efetuar um filtro pelo login.

Para cada registro de acesso de usuário, é possível rastrear os logs de SQL, logs de alterações de registros, e logs de requisições HTTP gerados por uma determinada sessão, por meio de botões que levam o administrador para outra tela contendo aqueles registros filtrados. Assim, ao clicar em “Log de SQL”, o administrador será direcionado para a tela de Logs de SQL que virá filtrada somente com os logs de SQL daquela sessão.

Access Log							
MARCOS ANTONIO RAFAEL DA FONSECA -							
Login							
<input type="text"/>							
<input type="button" value="Buscar"/>							
		id	sessionid	Login	login_time	logout_time	IP
<input type="button" value="Log de SQL"/>	<input type="button" value="Log de alterações"/>	<input type="button" value="Log de request"/>	24	95hpo3upnl9k6fg0dmot3kk6em	admin	2019-10-09 23:14:30	127.0.0.1
<input type="button" value="Log de SQL"/>	<input type="button" value="Log de alterações"/>	<input type="button" value="Log de request"/>	23	l5a4f7b64mrtfmip1qqso3obvrs	admin	2019-10-09 19:42:35	127.0.0.1
<input type="button" value="Log de SQL"/>	<input type="button" value="Log de alterações"/>	<input type="button" value="Log de request"/>	22	anl68bhahrflmbrurpk7m5ae9	admin	2019-10-09 00:14:59	127.0.0.1
<input type="button" value="Log de SQL"/>	<input type="button" value="Log de alterações"/>	<input type="button" value="Log de request"/>	21	3550pdulpfa8mdsd7buji8tbd3	admin	2019-10-08 23:47:41	2019-10-09 00:09:23
<input type="button" value="Log de SQL"/>	<input type="button" value="Log de alterações"/>	<input type="button" value="Log de request"/>	20	06r7u0v5tu0ratfvqa1e04cb5i	admin	2019-10-08 23:14:54	2019-10-08 23:15:06
<input type="button" value="Log de SQL"/>	<input type="button" value="Log de alterações"/>	<input type="button" value="Log de request"/>	19	pvo8sd1p5s37l624e195888sjt	user	2019-10-08 23:09:50	2019-10-08 23:09:56
<input type="button" value="Log de SQL"/>	<input type="button" value="Log de alterações"/>	<input type="button" value="Log de request"/>	18	ef8rh9t6t3o10a5ubg03n3lu	admin	2019-10-08 23:05:58	2019-10-08 23:09:47
<input type="button" value="Log de SQL"/>	<input type="button" value="Log de alterações"/>	<input type="button" value="Log de request"/>	17	nmde437rf5pt0td64vdsmf3gd	admin	2019-10-08 23:05:43	2019-10-08 23:05:55

Figura 157 Logs de acesso

6.4.5 Logs de SQL

O log de alterações de SQL precisa ser habilitado para cada base de dados que você deseja registrar os logs. Você pode querer registrar logs de SQL para uma base de dados, mas não para outra.

Para habilitar o log de instruções SQL, basta adicionar ao final do arquivo de configuração do banco de dados (INI), a definição `slog = SystemSqlLogService`. Basicamente esta definição (`slog` vem de *Standard Logger*) indica o nome de uma classe de log para receber as instruções geradas pelo sistema.

Você pode implementar a sua própria classe global de log, desde que ela implemente a interface `AdiantiLoggerInterface`, contendo o método `write($message)`. Se você mantiver a classe padrão `SystemSqlLogService`, automaticamente, registrará os comandos SQL na tabela `system_sql_log`.

app/config/sua-base-de-dados.ini

```
host  = <seu host>
port  = <porta>
name  = <nome do banco>
user  = <user>
pass  = <senha>
type  = <tipo>
slog  = SystemSqlLogService
```

A tela de consulta de logs de SQL pode ser acessada sob o menu “Logs”. Esta tela contém informações como a data, o login do usuário, o banco de dados e o comando SQL gerado, o nome do programa, a API, e o IP. Pode-se efetuar um filtro pelo login, banco de dados, instrução SQL, programa, sessão, ou requisição.

Ao posicionar o mouse sobre o registro, um quadro será exibido sobre a tela com todo o detalhamento do rastreio (trace) que gerou aquela instrução SQL, contendo nomes de arquivos e números de linhas.

The screenshot shows the 'SQL Log' interface. At the top, there are search fields for 'Login', 'Programa', 'Banco de dados', 'Sessão', 'SQL', and 'Requisição'. Below these is a 'Buscar' button. The main area displays a table of log entries:

ID	Data	Login	Banco de dados	SQL	Programa	SAPI	IP
1	2019-10-09 23:19:35	admin	permission	<pre><code>UPDATE system_program SET name = 'System Group Formx', controller = 'SystemGroupForm' WHERE (id = '1')</code></pre>	SystemProgramForm	apache2handler	127.0.0.1
2	2019-10-09 23:19:35	admin	permission	<pre><code>DELETE FROM system_group_program WHERE (system_group_id = '1')</code></pre>	SystemProgramForm	apache2handler	127.0.0.1

Below the table, a note reads: "Transaction: 5d9e9537361cf". The bottom right corner of the table area contains the text "MARCOS ANTONIO RAFAEL DA FONSECA -".

Figura 158 Logs de SQL

6.4.6 Logs de alterações

Para habilitar o log de mudanças de registros, você precisará incluir uma linha em cada classe de modelo (Active Record) que deseja registrar as alterações. Este log não é global, pois nem sempre desejamos registrar logs de alterações em todas as tabelas do sistema. Assim, sempre que quisermos registrar os logs de mudanças sobre uma classe, basta incluirmos a linha `use SystemChangeLogTrait`.

Esta linha inclui um Trait, que contém um conjunto de métodos que pode ser incluído em uma ou mais classes de modelo. Alguns chamam Traits de “método inteligente para copy & paste”, uma vez que o Trait evita a necessidade de copiar e colar um determinado trecho de código entre diferentes classes, facilitando o reaproveitamento de uma funcionalidade. Em seguida, vamos explicar o que faz este Trait.

app/model/admin/SystemProgram.class.php

```
<?php
class SystemProgram extends TRecord
{
    const TABLENAME = 'system_program';
    const PRIMARYKEY= 'id';
    const IDPOLICY = 'max';

    use SystemChangeLogTrait; // inclui os métodos de log de mudanças
}
```

A tela de consulta de logs de alteração pode ser acessada sob o menu “Logs”. Esta tela contém informações como a chave primária alterada, o momento da alteração, o login do usuário, a tabela, a coluna modificada, a operação realizada (criação, alteração, exclusão), o valor velho, e o novo, o programa que gerou a mudança, a API, e o IP. Pode-se efetuar um filtro por tabela, login, programa, ou sessão. Os registros são agrupados por transação de banco de dados.

Ao posicionar o mouse sobre o registro, um quadro será exibido sobre a tela com todo o detalhamento do rastreio (trace) que gerou aquela instrução SQL, contendo nomes de arquivos e números de linhas.

Table change log																							
MARcos ANTONIO RAFAEL DA FONSECA -																							
Data	Login	Tabela	PK	Coluna	Operação	Valor antigo	Valor novo	Programa	SAPI	IP													
Transaction: 5d9e9537361cf																							
2019-10-09 23:19:35	admin	system_program	1	name	changed	System Group Form	System Group Formx	SystemProgramForm	apache2handler	127.0.0.1													
Transaction: 5d9e9539757a3																							
2019-10-09 23:19:37	admin	system_program	1	name	changed	System Group Formx	System Group Form	SystemProgramForm	apache2handler	127.0.0.1													
1 a 2 de 2 registros																							
<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td></td><td></td></tr> </table>												1	2	3	4	5	6	7	8	9	10		
1	2	3	4	5	6	7	8	9	10														
MARcos ANTONIO RAFAEL DA FONSECA -																							

Figura 159 Logs de alteração de registros

Um Trait contém um conjunto de métodos que pode ser importado em uma ou várias classes. O Trait `SystemChangeLogTrait` contém os métodos como `onAfterDelete()`, `onBeforeStore()`, e `onAfterStore()`. Estes métodos, ao serem inseridos em uma classe, alteram o comportamento padrão e inserem ações automaticamente no momento de salvar e excluir registros. Internamente, estes métodos utilizam a classe `SystemChangeLogService` para registrar as mudanças.

app/model/log/SystemChangeLogTrait.php

```
<?php
trait SystemChangeLogTrait
{
    public function onAfterDelete( $object )
    {
        SystemChangeLogService::register($this, $object, array());
    }

    public function onBeforeStore($object)
    {
        $pk = $this->getPrimaryKey();
        $this->lastState = array();
        if (isset($object->$pk) and self::exists($object->$pk)) {
            $this->lastState = parent::load($object->$pk)->toArray();
        }
    }
    public function onAfterStore($object)
    {
        SystemChangeLogService::register($this, $this->lastState, (array) $object);
    }
}
```

6.4.7 Logs de requisição

Os logs de requisição registram cada uma das requisições HTTP que chegam à aplicação. Requisições derivadas da navegação, a partir de chamadas REST, e até mesmo a partir de chamadas pela linha de comando são registradas. Na tela a seguir podemos verificar o registro de logs de requisição, acessível a partir do menu Logs. Para cada registro de Log de requisição gerado, temos muitas informações relacionadas, como o IP, o login, a sessão, o endereço, os cabeçalhos e o conteúdo da requisição, dentre outros.

ID	Hora	sessionid	View	SQL
6	2019-10-09 23:20:35	95hpo3upnl9kf0dmot3k	View	SQL
5	2019-10-09 23:20:34	95hpo3upnl9kf0dmot3k	View	SQL
4	2019-10-09 23:20:32	95hpo3upnl9kf0dmot3k	View	SQL
3	2019-10-09 23:20:30	95hpo3upnl9kf0dmot3k	View	SQL
2	2019-10-09 23:20:28	95hpo3upnl9kf0dmot3k	View	SQL
1	2019-10-09 23:20:27	95hpo3upnl9kf0dmot3k	View	SQL

Figura 160 Logs de requisição

Para habilitar o registro de requisições, é necessário alterar o `application.ini` e ligar a variável `request_log`. Também é necessário indicar na variável `request_log_service` o nome da classe de serviço que irá registrar os logs de request. A classe `SystemRequestLogService` é a classe nativa que registra esses logs por meio da classe de modelo `SystemRequestLog` na tabela `system_request_log`. A variável `request_log_types`, permite definir exatamente quais tipos de requisição serão registradas: `cli` para linha de comando, `web` para navegação Web, e `rest` para Rest services.

app/config/application.ini

```
[general]
request_log = 1
request_log_service = SystemRequestLogService
request_log_types = cli,web,rest
```

6.5 Módulo Comunicação

O modulo de comunicação permite o compartilhamento de documentos e mensagens (inbox) entre os usuários, bem como registro de notificações do sistema.

Nesta seção abordaremos o módulo de Comunicação, com sua modelagem de classes, tabelas, e funcionalidades.

6.5.1 Diagrama de classes

Na próxima figura, temos o diagrama de classes do módulo de comunicação, com as classes desta camada (`app/model/communication/`). A seguir, temos uma lista com as classes Active Record:

- SystemDocumentCategory**: Representa uma categoria (tipo) de documento. Ex: documentação, especificação, manuais, etc;
- SystemDocument**: Representa um documento a ser compartilhado. Possui agregação com `SystemUser` e `SystemGroup`, uma vez que o documento pode ser compartilhado com muitos usuários e grupos de usuários;
- SystemMessage**: Representa uma mensagem enviada de um usuário para outro, em um determinado momento;
- SystemNotification**: Representa uma notificação gerada pelo sistema, tendo como destinatário um usuário, que é solicitado a realizar uma ação;

A figura seguir, demonstra as principais classes do módulo de comunicação. Essas classes são armazenadas sob o diretório `app/model/communication` do Template.

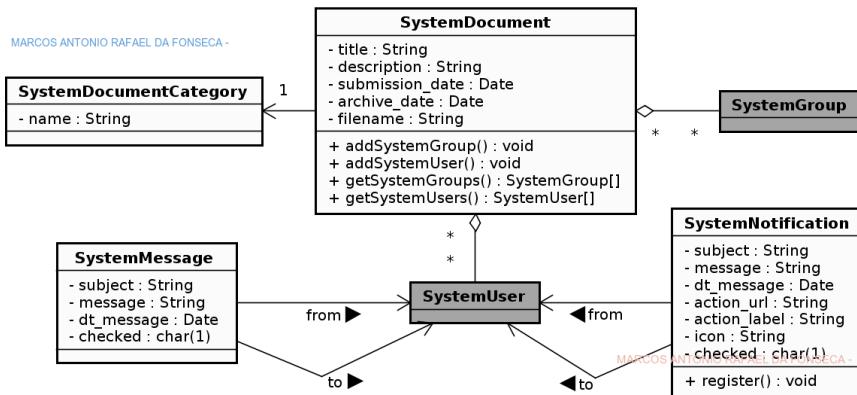


Figura 161 Diagrama de classes de comunicação

Obs: As classes **SystemUser** e **SystemGroup** pertencem a outro modelo, por isso são representadas por cores diferentes.

6.5.2 Modelo relacional

As classes de comunicação manipulam dados que são registrados em tabelas do banco communication. Esse banco de dados é configurado no arquivo `app/config/communication.ini`. A instalação padrão do Template acompanha um banco de dados no formato SQLite, (`app/database/communication.db`). Porém, neste mesmo diretório encontram-se as instruções SQL (`communication.sql`) para criação do banco de comunicação em outros sistemas de bancos de dados. Quando utilizar outro banco de dados, você precisará alterar a configuração de acesso ao banco de dados. A seguir, o arquivo de configuração de acesso ao banco de comunicação. Para reunir este modelo na mesma base da aplicação, basta importar os dois SQL's no mesmo banco de dados, e fazer os dois INI's apontarem para a mesma base.

Obs: Não exclua sob hipótese alguma o arquivo `communication.ini`. Mesmo que você decida reunir a base de comunicação com a base da aplicação, mantenha os diferentes INI's apontando para a mesma base de dados, uma vez que existem vários pontos dentro do template que referenciam este arquivo no momento de abrir conexão com a base de dados.

`app/config/communication.ini`

```
host  =
port  =
name  = app/database/communication.db
user  =
pass  =
type  = sqlite
```

A seguir, estão as tabelas de comunicação. Basicamente, temos as tabelas:

- system_document_category**: armazena uma categoria de documento. Uma categoria pode ser, por exemplo: manual, especificação, regulamentos;
- system_document**: armazena a referência para um documento e seus metadados, como título, referência para a categoria, data, dentre outros;
- system_document_group**: armazena a agregação entre documentos e grupos, ou seja, os vínculos de quais grupos podem acessar aquele documento;
- system_document_user**: armazena a agregação entre documentos e usuários, ou seja, os vínculos de quais usuários podem acessar aquele documento;
- system_message**: armazena uma mensagem enviada entre usuários. Uma mensagem contém, dentre outros, título, conteúdo, remetente, destinatário;
- system_notification**: armazena uma notificação de sistema. Uma notificação contém uma mensagem, e esta pode estar associada a uma ação a ser realizada pelo destinatário. Ex: aprovar um pedido.

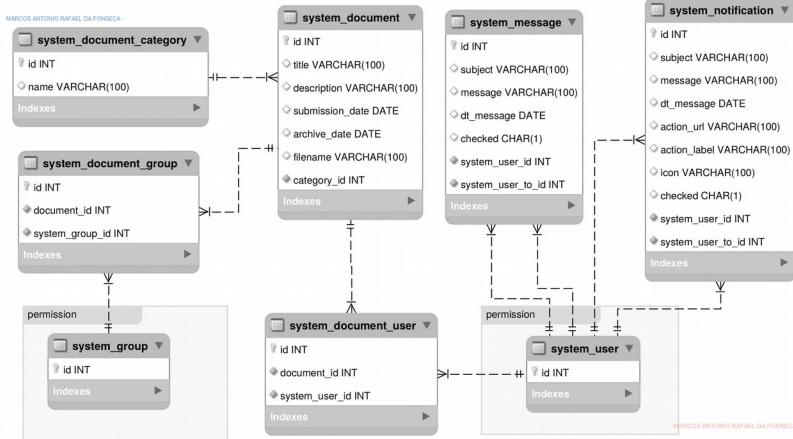


Figura 162 Tabelas de comunicação

6.5.3 Gestão de documentos

Uma das principais funcionalidades do módulo de comunicação é a gestão de documentos. Neste módulo, um usuário pode realizar upload de um documento qualquer (html, pdf, zip, rtf, csv, doc, docx, xls, xlsx, ppt, pptx, odt, ods, odp, ...) por meio do link “Enviar documento” e definir alguns atributos, como título, descrição, uma data de envio, uma categoria, além de poder compartilhar o documento com demais usuários e grupos de usuários para que estes possam consultá-lo. A figura a seguir exibe a tela de propriedades do documento, bem como compartilhamento.

Documento

MARCOS ANTONIO RAFAEL DA FONSECA -

Título	Manual do usuário
Descrição	Contém o manual do sistema
Categoria	Documentação
Data de submissão	2019-10-10 <input type="button" value=""/>
Data de arquivamento	<input type="button" value=""/>

Permissão

Usuários	x User
Grupos	<input checked="" type="checkbox"/> Admin <input type="checkbox"/> Standard

MARCOS ANTONIO RAFAEL DA FONSECA ~

Figura 163 Propriedades do documento

Os documentos são exibidos em uma listagem de documentos do usuário, acessada por meio do link “Meus documentos”. O dono do documento (quem submeteu) pode a qualquer momento baixar o documento, substituir o documento por outro, editar suas propriedades, ou excluí-lo. Ao editar, o usuário pode definir uma data de arquivamento para o arquivo, sendo que o documento já deixa de ser exibido para os demais usuários. A figura a seguir mostra a listagem de documentos do usuário.

Documentos

MARCOS ANTONIO RAFAEL DA FONSECA -

Título	<input type="text"/>
Categoria	<input type="button" value=""/>
Arquivo	<input type="button" value=""/>

Id	Título	Categoria	Data	Usuário
1	Amostra	Documentação	2019-09-21	Administrator

1 a 1 de 1 registros

MARCOS ANTONIO RAFAEL DA FONSECA ~

Figura 164 Listagem de documentos do usuário

O usuário final que está associado a um documento por meio de seu login, ou por meio de seu grupo de usuários, poderá somente consultar o documento por meio do link “Compartilhados comigo”, que possui uma listagem similar, porém somente com o botão para realizar o download do arquivo.

6.5.4 Troca de mensagens

A funcionalidade de troca de mensagens permite aos usuários enviarem mensagens entre si por meio da aplicação. Em um menu *dropdown* localizado ao lado da caixa de busca de programas, o usuário tem acesso às mensagens ainda não lidas, bem como à função para ler mensagens, e enviar novas mensagens.

Ao acessar uma mensagem, ou a função para “Ler mensagens”, o usuário terá acesso a uma interface similar à de uma ferramenta de e-mails, com funções de compor nova mensagem, ver a caixa de entrada, os itens enviados, e os arquivados.

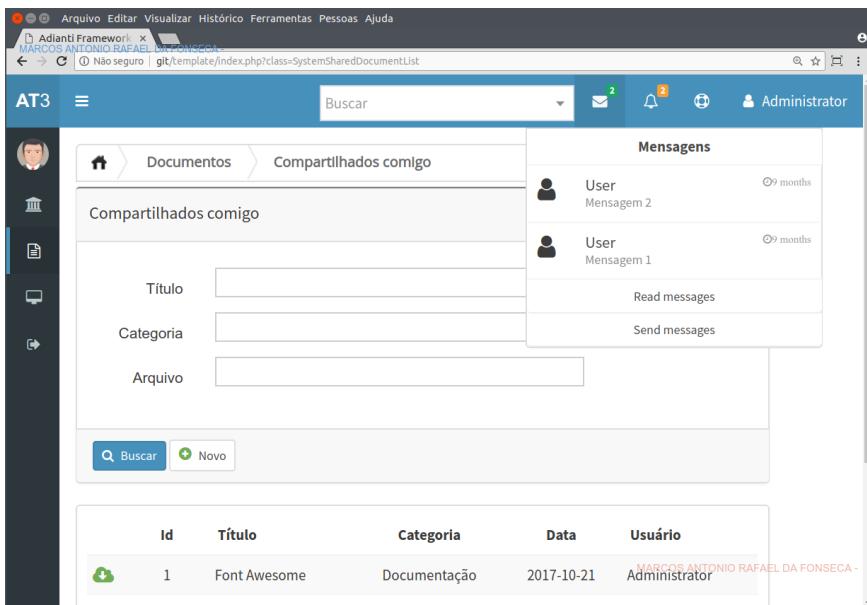


Figura 165 Menu de mensagens

6.5.5 Notificações do sistema

Existem situações em que a aplicação precisa alertar os usuários sobre um evento. Podemos citar como exemplo, em um sistema empresarial, quando um pedido chega à fase de aguardar aprovação. Neste caso, o sistema pode enviar uma notificação para que o aprovador tome a ação de aprovar o que está pendente. Assim, o aprovador recebe a notificação e toma uma ação que o leva a um certo endereço, uma URL.

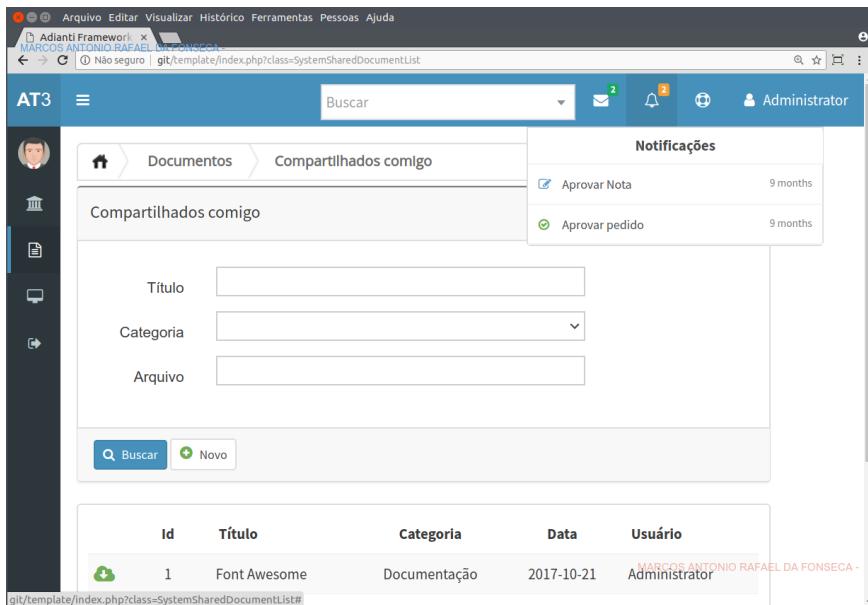
No Framework, é possível gerar notificações via sistema em qualquer parte do código-fonte por meio do método `SystemNotification::register()`. Este método recebe como parâmetros o ID do usuário de destino da notificação, o assunto, a mensagem, uma ação (URL), um label (rótulo) para o botão de ação, e um ícone.

As notificações podem ser vistas no canto superior direito da tela, logo após o envelope de troca de mensagens. Assim que o usuário clicar sobre a notificação, a mesma será apresentada em tela. O usuário verá o conteúdo da notificação e poderá clicar no botão. Assim que clicar no botão, será direcionado para a ação (URL) programada.

Exemplo de notificação

```
<?php
```

```
SystemNotification::register( 2, // usuario destino  
    'Documento novo', // assunto  
    'Leia a nova documentação', // mensagem  
    'class=SystemSharedDocumentList', // acao do botão  
    'Ver documentos', // label do botão  
    'fa fa-list-alt green' ); // ícone do botão
```



The screenshot shows the AT3 application interface. At the top, there's a navigation bar with links like Arquivo, Editar, Visualizar, Histórico, Ferramentas, Pessoas, and Ajuda. Below that is a header bar with the title 'AT3', a search bar, and notification icons (2 messages, 1 notification). On the left, there's a sidebar with user profile information ('MARCOS ANTONIO RAFAEL DA FONSECA') and a navigation menu with icons for Home, Documents, Shared, and others. The main content area shows a 'Compartilhados comigo' (Shared with me) section with fields for Título, Categoria, and Arquivo. To the right, there's a 'Notificações' (Notifications) panel listing two items: 'Aprovar Nota' (Approved Note) and 'Aprovar pedido' (Approved Request), both from 9 months ago. At the bottom, there's a table with columns Id, Título, Categoria, Data, and Usuário, showing one row with Id 1, Título 'Font Awesome', Categoria 'Documentação', Data '2017-10-21', and Usuário 'MARCOS ANTONIO RAFAEL DA FONSECA'.

Figura 166 Notificações do sistema

6.6 Dicas de utilização

6.6.1 Novos temas

Os temas distribuídos junto com o Template precisam passar por alguns critérios, como ser de utilização livre, ou seja, a licença precisa permitir a sua redistribuição. Mas cada desenvolvedor é livre para comprar novos temas e integrá-los na aplicação. Para utilizar um novo tema no Template, é importante identificar os elementos que são diferentes de um tema para outro. Dentre eles, podemos citar:

- Layout:** utilizam um layout totalmente diferente (`app/templates/themeX`);
- Index:** a forma de montagem do menu, que é realizada pela classe `AdiantiMenuBuilder` é diferente (`index.php`).

Layout

Para criar um tema novo, inicialmente deve ser criado um subdiretório em `app/templates`. No diretório criado, devem ser colocados todos os arquivos específicos do tema (CSS, JS, HTML). Você precisará pelo menos dos arquivos de layout: `layout.html`, `public.html` e `login.html`, além de uma cópia do `libraries.html`. O arquivo de layout deve ter as marcações `{LIBRARIES}` e `{HEAD}`, um `<div>` com o ID `adianti_div_content`, outro, com ID `adianti_online_content`, `adianti_online_content` e `adianti_right_panel`. Esses elementos contêm o que for renderizado e já foram explicados anteriormente. Para ativar o tema, basta alterar a variável `theme`, no arquivo `application.ini`.

`app/config/application.ini`

```
[general]
language = pt
application = template
theme = theme3
```

Obs: Evite importar outras cópias de bibliotecas como jQuery e Bootstrap, que já são disponibilizadas e homologadas pelo Framework, para evitar conflitos.

Index

No arquivo `index.php`, existe uma chamada para o método `parse()` da classe `AdiantiMenuBuilder`. Este método identifica o tema passado como parâmetro e utiliza um certo método de construção do menu, adaptado para o tema escolhido. Existem diferentes métodos de montagem de menu. Dentro da classe `AdiantiMenuBuilder` você verá um bloco switch/case. Para cada template, diferentes classes CSS são utilizadas para a montagem do menu.

`index.php`

```
<?php
$menu_string = AdiantiMenuBuilder::parse('menu.xml', $theme);
```

6.6.2 Alterando as cores do tema 3

Os temas distribuídos com o template podem ter personalização de cores. O tema 3, por exemplo, que é baseado no tema AdminLTE, possui skins. Cada skin traz uma temática de cores em sua versão clara e escura.

Os skins disponíveis por padrão de fábrica para o tema 3 são: `skin-black`, `skin-black-light`, `skin-blue`, `skin-blue-light`, `skin-green`, `skin-green-light`, `skin-purple`, `skin-purple-light`, `skin-red`, `skin-red-light`, `skin-yellow`, `skin-yellow-light`. Para utilizar um desses skins, basta inserir seu nome na classe da tag `body` do layout.

Outra classe CSS bastante útil é a `layout-boxed`. Esta classe torna o layout “encaixotado”, ou seja, reduzido à uma largura fixa, não expandido até completar a largura da tela, como pode ser visto na figura a seguir.

A seguir, temos um exemplo de combinação do skin `skin-purple-light`, que traz a tela em roxo com menu claro, e o layout do tipo largura fixa `layout-boxed`.

app/templates/theme3/layout.html

```
<body class="hold-transition skin-purple-light layout-boxed sidebar-mini">
```

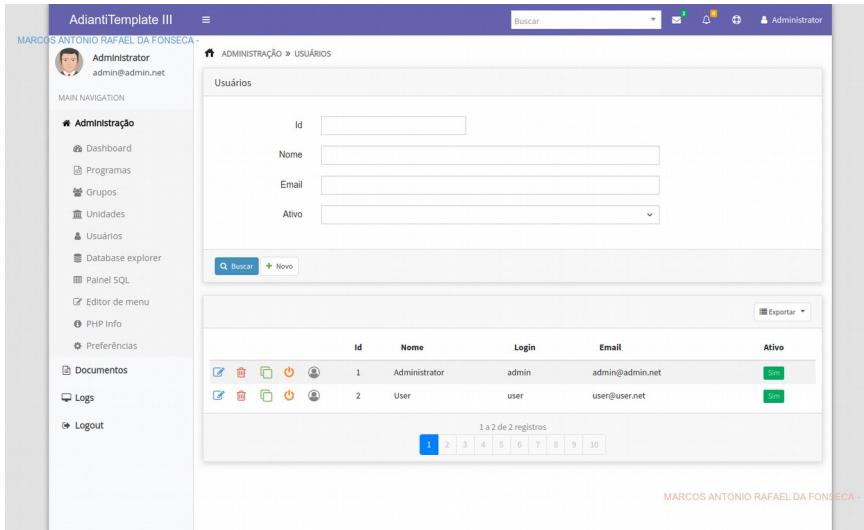


Figura 167 Tema 3 com layout customizado

6.6.3 Alterando as cores do tema 4

O tema 4, baseado no template AdminBSB também oferece temas de cores. Atualmente os temas disponíveis de fábrica são: theme-amber, theme-black, theme-blue, theme-blue-grey, theme-brown, theme-cyan, theme-deep-orange, theme-deep-purple, theme-green, theme-grey, theme-indigo, theme-light-blue, theme-lime, theme-orange, theme-pink, theme-purple, theme-red, theme-teal, theme-yellow.

Para aplicar um tema, basta alterar o arquivo `layout.html`, e editar a classe do body, inserindo o nome do tema, como demonstrado a seguir. A próxima figura exibe o tema 4 com o tema `theme-deep-purple` aplicado.

`app/templates/theme4/layout.html`

```
<body class="theme-deep-purple">
```

The screenshot shows the Adianti Template IV application interface. The top navigation bar is purple, featuring the title "Adianti Template IV" and the user "MARCOS ANTONIO RAFAEL DA FONSECA". Below the header is a sidebar with a user profile picture and the text "Administrator" and "admin@admin.net". The sidebar also contains a "MENU" section with links for "Administração", "Documentos", "Logs", and "Logout". The main content area has a title "ADMINISTRAÇÃO > USUÁRIOS". It contains a search form with fields for "Id", "Nome", "Email", and "Ativo", along with "Buscar" and "Novo" buttons. Below the form is a table with two rows of data:

				1	Administrator	admin	admin@admin.net	Sim
				2	User	user	user@user.net	Sim

At the bottom of the page, there is a footer with the text "MARCOS ANTONIO RAFAEL DA FONSECA".

Figura 168 Tema 4 com layout customizado

6.6.4 Criando um programa dentro do Template

Para criar um programa e disponibilizá-lo dentro da estrutura do Template, é necessário seguir alguns passos. Estes passos envolvem a liberação de permissão de acesso ao novo programa, bem como a sua disponibilização no menu:

- ① **Instalar o template:** descompactar o Template para criação de sistemas (novo projeto);
- ② **Criar o programa:** implementar o novo programa (Ex: `TesteForm`) e salvá-lo sob a estrutura de `app/control`. Sugestão: inicialmente implementar somente a declaração da classe;
- ③ **Cadastrar o programa:** logar como admin, e cadastrar o novo programa em Administration → Programs (Preencher no campo Controller, o nome da classe criada Ex: `TesteForm`) e no campo nome, fornecer um nome legível;
- ④ **Conceder permissão:** conceder permissão de acesso ao novo programa para um usuário (Administration → Users) ou para um grupo (Administration → Groups);
- ⑤ **Acrescentar no menu:** acrescentar este programa no `menu.xml`, que contém a estrutura hierárquica do menu do sistema, ou usar o editor de menu para tal;
- ⑥ **Recarregar permissões:** efetuar logout e login novamente com um usuário que tenha permissão de acesso ao programa, ou clicar no avatar do usuário, e em seguida em recarregar;
- ⑦ **Terminar:** terminar de desenvolver o programa.

Obs: Sempre que uma nova permissão for cadastrada, é importante efetuar um novo login no sistema, pois é no login que as permissões são carregadas na sessão.

6.6.5 Práticas responsivas

Para que os formulários e datagrids construídos dentro do Template tenham características responsivas mediante o redimensionamento em tela, é necessário seguir algumas diretrizes no momento de criar as telas. A seguir, veremos alguns cuidados na criação de formulários e datagrids.

Para formulários

A classe `BootstrapFormBuilder` é a mais indicada para criação de formulários, pois constrói a interface utilizando uma estrutura de HTML de acordo com a biblioteca Bootstrap, sendo naturalmente responsiva. Assim, a disposição dos campos se ajustará automaticamente ao tamanho da tela.

Além disso, é recomendado utilizar tamanhos em percentual no método `setSize()`, e não tamanhos fixos. Use por exemplo `setSize('80%')`, e não `setSize(200)`.

Exemplo de formulário responsável

```
<?php
class TesteForm extends TPage
{
    protected $form;

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

        // cria o formulário
        $this->form = new BootstrapFormBuilder('form_teste');
        $this->form->setFormTitle( 'Teste' );

        // cria os campos
        $id = new TEntry('id');
        $name = new TEntry('name');

        // adiciona campos ao formulário
        $this->form->addFields( [new TLabel('Id')], [$id] );
        $this->form->addFields( [new TLabel(_t('Name'))], [$name] );

        // define tamanho dos campos
        $id->setSize('30%');
        $name->setSize('70%');

        // adiciona o formulário à página
        parent::__add($this->form);
    }
}
```

Para datagrids

Ao criarmos datagrids, também devemos tomar alguns cuidados na construção da interface. Ao criarmos as colunas dadatagrid (`TGridColumn`), não devemos especificar um tamanho absoluto (quarto parâmetro do construtor), como pode ser visto no trecho em destaque, mas poderemos utilizar um tamanho em percentual.

Além disso, adatagrid deve ter largura de 100% (`$this->datagrid->style = 'width: 100%'`) para ocupar todo o espaço disponível dentro de seu container, que por sua vez, deve ter uma largura definida em termos percentuais (Ex: 80%). Também é sugerida a utilização da classe `BootstrapDatagridWrapper`.

Exemplo dedatagrid responsiva

```
<?php
class ResponsiveList extends TPage
{
    private $datagrid;
    public function __construct()
    {
        parent::__construct();

        $this->datagrid = new BootstrapDatagridWrapper(new Tdatagrid);
        $this->datagrid->style = 'width: 100%';

        $id    = new TGridColumn('id', 'ID', 'right');
        $name  = new TGridColumn('name', _t('Name'), 'left');

        $this->datagrid->addColumn($id);
        $this->datagrid->addColumn($name);

        $container = new TVBox;
        $container->style = 'width: 80%';
        $container->add($this->datagrid);
        parent::add($container);
    }
}
```

DataTables

As práticas que acabamos de apresentar são adequadas na maioria dos casos. Porém, em algumas situações, como em datagrids com muitas colunas, precisamos utilizar outras abordagens. Em datagrids com muitas colunas, mesmo que ela se adapte à largura da tela no redimensionamento, em algum momento o conteúdo dela estabelecerá uma largura mínima, e será criada uma barra de rolagem horizontal.

Para resolver este tipo de situação, existe a biblioteca DataTables. Esta biblioteca permite que as datagrids tenham um comportamento ainda mais dinâmico no redimensionamento. Quando umadatagrid possui o recurso de DataTables habilitado, e ocorre um grande redimensionamento (a área visível torna-se menor do que o conteúdo dadatagrid a ser exibido), então as colunas que não teriam espaço para exibição são escondidas e podem ser visualizadas sob demanda por meio de um botão (+), disponibilizado no início de cada linha.

PK	Data	Login	Tabela	Coluna	Operação
1	2018-08-18 18:24:38	admin	system_program	name	changed

Valor antigo : System Group Form

Valor novo : System Group Form

(+)	1	2018-08-18 18:24:40	admin	system_program	MARCOS ANTONIO RAFAEL DA FONSECA - changed
-----	---	------------------------	-------	----------------	--

Figura 169 Exemplo de uso de DataTables

Para habilitar o recurso de DataTables é bastante simples. Basta definir o atributo da datagrid `datatable = 'true'`. Você deve fazer isso para cada datagrid em que deseja que este recurso esteja disponível.

Exemplo dedatagrid com DataTables

```
$this->datagrid = new TDataGrid;
$this->datagrid->style = 'width: 100%';
$this->datagrid->datatable = 'true'; // habilita DataTables
```

Outra maneira de apresentar datagrids com muitas colunas é utilizar a rolagem horizontal. Para habilitar esta abordagem, é necessário definir uma largura para a datagrid, como neste exemplo (1600px), e adicionar a datagrid em um container como um `TPanelGroup` com o estilo `overflow-x:auto`, para gerar uma rolagem horizontal.

Exemplo dedatagrid com rolagem horizontal

```
$this->datagrid = new BootstrapDatagridWrapper(new TQuickGrid);
$this->datagrid->style = 'width: 1600px';

$panel = new TPanelGroup(_t('Horizontal Scrollable Datagrids'));
$panel->add($this->datagrid);
$panel->addFooter('footer');

$panel->getBody()->style = "overflow-x:auto;";
```

A figura a seguir apresenta um exemplo dedatagrid com rolagem horizontal.

The screenshot shows a web application interface titled 'AT5.5'. The top navigation bar includes search fields for 'Busca programa' and 'Busca código', and a user profile placeholder '{username}'. On the left, there's a vertical sidebar with icons for home, presentation, datagrids, and other system functions. The main content area has a breadcrumb navigation: 'Apresentação > Datagrids > Datagrid com scroll horizontal'. Below this is a title 'Datagrid com scroll horizontal'. A horizontal scroll bar is visible under a table containing four rows of data:

	Code	Name	Address	Phone
	1	Fábio Locatelli	Rua Expedicionario	1111-1111
	2	Julia Haubert	Rua Expedicionarios	2222-2222
	3	Carlos Ranzi	Rua Oliveira	3333-3333
	4	Daline DallOglio	Rua Oliveira	4444-4444

The bottom right corner of the table area contains the text 'MARCOS ANTONIO RAFAEL DA FONSECA -'. At the very bottom of the page is a footer section labeled 'footer'.

Figura 170 Datagrid com rolagem horizontal

Outra prática que permite exibir mais informações em datagrids sem precisar utilizar muitas colunas, é o popover. O popover, como já visto anteriormente, é um balão exibido quando o usuário passa o mouse sobre o registro. Podemos utilizar este balão para exibir diversas informações relacionadas ao registro.

Exemplo dedatagrid com popover

```
$this->datagrid = new BootstrapDatagridWrapper(new TQuickGrid);
$this->datagrid->enablePopover('Item details', '<table> ...{$var} </table>');
```

The screenshot shows a web application interface titled 'AT5.5'. The top navigation bar includes search fields for 'Busca programa' and 'Busca código', and a user profile placeholder '{username}'. On the left, there's a vertical sidebar with icons for home, presentation, datagrids, and other system functions. The main content area has a breadcrumb navigation: 'Apresentação > Datagrids > Datagrid com popover'. Below this is a title 'Datagrid com popover'. A popover window titled 'Item details' is displayed over the third row of the datagrid, showing detailed information for the item with code 3:

Code	Name	Item details		Phone
1	Fábio Locatelli	Name	Carlos Ranzi	1111-1111
2	Julia Haubert	Address	Rua Oliveira	2222-2222
3	Carlos Ranzi			3333-3333
4	Daline DallOglio			4444-4444

The bottom right corner of the table area contains the text 'MARCOS ANTONIO RAFAEL DA FONSECA -'. At the very bottom of the page is a footer section labeled 'footer'.

Figura 171 Datagrid com popover

CAPÍTULO 7

Estudos de caso

Após vários capítulos estudando o framework, já nos sentimos preparados para criar as nossas próprias aplicações, uma vez que já vimos como manipular entidades da base de dados, criar uma interface orientada a objetos, e também vimos que o Template nos oferece recursos essenciais na criação de novas aplicações, como: login, controle de permissões, logs, e comunicação, dentre outros. O objetivo deste capítulo é conhecer dois exemplos de aplicações construídas com o Adianti Framework.

7.1 Aplicação Library

A primeira aplicação que vamos estudar, chama-se Library (biblioteca). O objetivo desta aplicação é fornecer os controles mínimos que uma biblioteca precisa para funcionar, sempre visando a demonstração do framework. Esta aplicação não tem como objetivo ser um sistema completo para gerenciamento de bibliotecas, uma vez que já existem inúmeros softwares com esta finalidade.

A aplicação Library fornece algumas funcionalidades como a catalogação de livros, a classificação de livros, cadastro de leitores, autores, assuntos e editoras, empréstimo e devolução de livros, relatórios de livros, leitores e empréstimos, dentre outros.

A aplicação Library é baseada no Template e já herda deste o controle de permissão, compartilhamento de documentos, e também a comunicação entre usuários por trocas de mensagens. Esta aplicação possui diferentes perfis de usuários, sendo que cada perfil possui acesso a um grupo de funcionalidades do sistema. Poderemos ver melhor as funcionalidades acessadas por cada um dos atores, no diagrama de casos de uso. Ao longo deste capítulo, abordaremos como a aplicação está estruturada.

Obs: Baixe o Library no endereço <http://www.adianti.com.br/framework-library>.

7.1.1 Conteúdo da aplicação

O Library é construído sobre a estrutura do Template, e oferece o seguinte conteúdo específico e adicional sobre esta estrutura:

Tabela 10. Conteúdos do Library

app	Diretório da aplicação
config library.ini	Pasta com configurações. Configuração de acesso para a base biblioteca.
control library	Classes de controle. Classes de controle de biblioteca.
database library.db library.sql	Arquivos de base de dados em formato SQLite. Base de dados em SQLite para biblioteca. Script de criação da base da biblioteca.
model library	Classes de modelo (Active Records). Classes de modelo de biblioteca.

7.1.2 Diagrama de classes

Na próxima figura, temos o diagrama de classes do sistema, com as classes da camada de modelo (app/model).

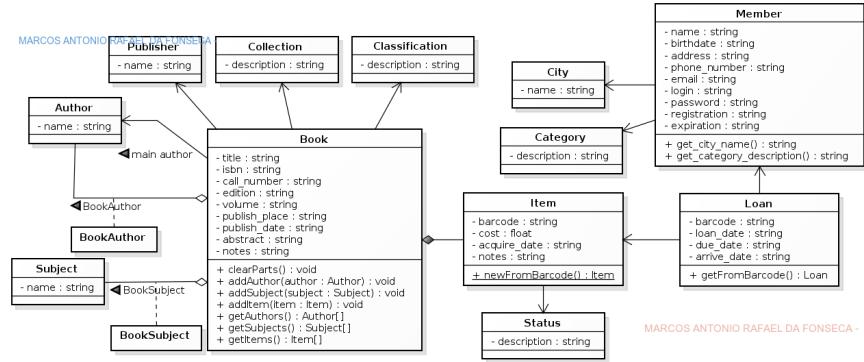


Figura 172 Diagrama de classes da biblioteca

A seguir, temos uma lista com a descrição das classes Active Record:

- Author**: representa um autor de uma obra (Ex: Machado de Assis);
- Subject**: representa um assunto de uma obra (Ex: Informática);
- Publisher**: representa uma editora responsável pela publicação da obra;
- Collection**: representa uma coleção (livro, periódico, dvd, dicionário);
- Classification**: representa uma classificação (música, medicina, ciência);
- Book**: representa uma obra com título, isbn, edição, etc.. Possui associação com Publisher, Collection e Classification. Possui agregação com Author e Subject, uma vez que um livro pode ter vários autores e assuntos, e estes podem

estar presentes em obras diferentes. Possui composição com `Item`, que representa cada um dos exemplares físicos da obra na biblioteca;

- `Item`: representa um exemplar de uma obra. Cada exemplar possui o seu próprio código de barras, custo, data de aquisição. Associado com `Status`;
- `Status`: representa o estado de um livro (emprestado, disponível, etc.);
- `Loan`: representa o empréstimo de um livro com data de empréstimo, data prevista, data de devolução, etc.. Possui uma associação com `Item` e `Member`;
- `Member`: representa um membro da biblioteca. Possui associação com `City` e `Category`;
- `City`: representa uma cidade, na qual o usuário da biblioteca reside;
- `Category`: representa uma categoria de usuário (aluno, professor, etc.).

7.1.3 Modelo relacional

O modelo relacional, que pode ser visto na figura após a tabela a seguir, pode ser derivado a partir do modelo de classes persistentes, por meio de técnicas de mapeamento objeto-relacional. No modelo relacional do sistema Library, temos as tabelas a seguir. Juntamente a cada tabela, temos a explicação sobre seu mapeamento:

- `author`: armazena um Active Record do tipo `Author`;
- `subject`: armazena um Active Record do tipo `Subject`;
- `publisher`: armazena um Active Record do tipo `Publisher`;
- `collection`: armazena um Active Record do tipo `Collection`;
- `classification`: armazena um Active Record do tipo `Classification`;
- `book`: armazena um Active Record `Book`. A associação de `Book` com `Publisher` foi mapeada por meio da chave estrangeira `publisher_id`. Sua associação com `Collection` foi mapeada por meio da chave estrangeira `collection_id`. Sua associação com `Classification` foi mapeada por meio da chave estrangeira `classification_id`. Sua agregação com `Author` foi mapeada por meio da tabela associativa `book_author`. Sua agregação com `Subject` foi mapeada por meio da tabela associativa `book_subject`;
- `item`: armazena um Active Record `Item`. A composição de `Book` com `Item` foi representada por meio da chave estrangeira `book_id`. Sua associação com `Status` foi mapeada por meio da chave estrangeira `status_id`;
- `status`: armazena um Active Record `Status`.
- `loan`: armazena um Active Record `Loan`. Sua associação com `Member` foi mapeada por meio da chave estrangeira `member_id`;
- `member`: armazena um Active Record `Member`. Sua associação com `Category` foi mapeada por meio da chave estrangeira `category_id`. Sua associação com `City` foi mapeada por meio da chave estrangeira `city_id`;
- `city`: armazena um Active Record `City`;
- `category`: armazena um Active Record `Category`.

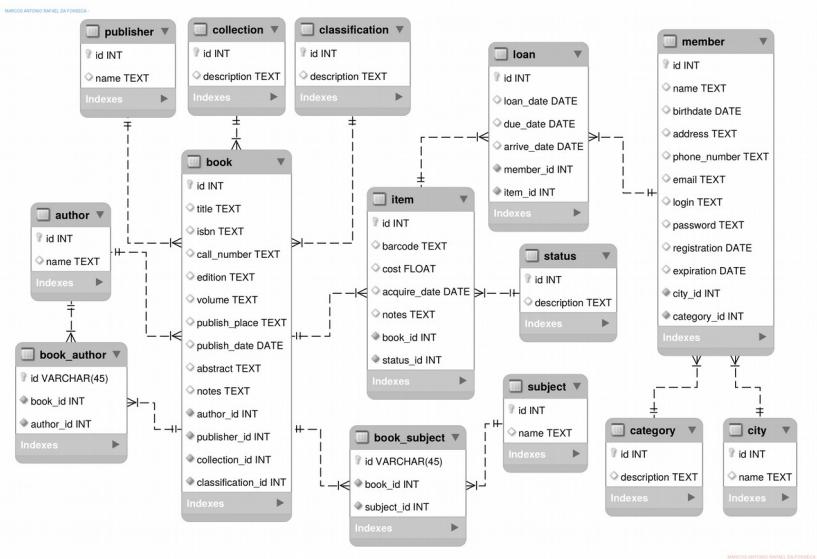


Figura 173 Modelo relacional da biblioteca

7.1.4 Diagrama de casos de uso

O diagrama de casos de uso a seguir, procura demonstrar os principais papéis de usuários no sistema, bem como as suas responsabilidades.

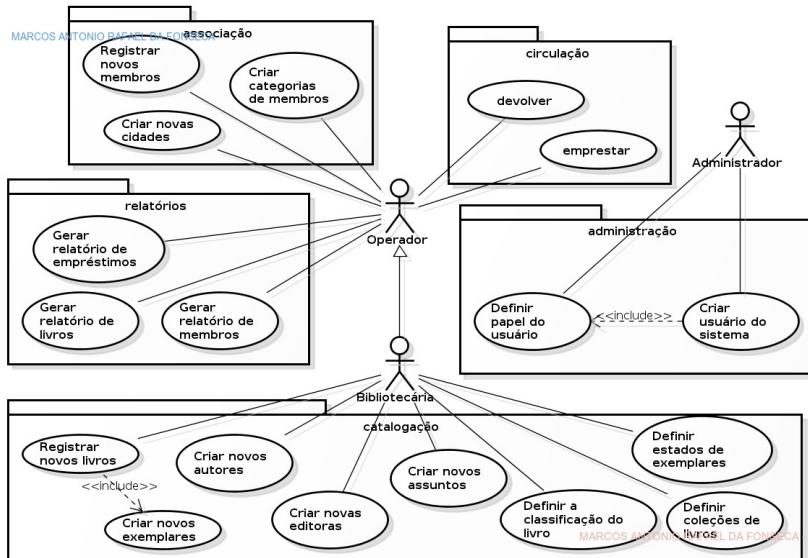


Figura 174 Diagrama de casos de uso da biblioteca

O operador é o funcionário responsável pelas tarefas cotidianas como empréstimos (grupo circulação), e cadastro de novos membros (grupo associação). A Bibliotecária é um tipo de operador. Isto significa que ela herda os casos de uso de operador e também possui operações exclusivas, como o registro de novos livros.

7.1.5 Especificação dos casos de uso

A tabela a seguir procura explicar de maneira mais completa as funcionalidades que cada papel poderá acessar dentro do sistema.

Tabela 11. Papéis e responsabilidades para a aplicação Library

Caso de uso	Papel	Descrição
Registrar novos membros	Operador	Registra novos membros na biblioteca (nome, endereço, telefone, etc.).
Criar categorias de membros	Operador	Cria novas categorias (aluno, funcionário, professor) para os membros da biblioteca.
Criar cidades	Operador	Cria novas cidades para usar no cadastro de membros.
Gerar relatório de empréstimos	Operador	Gera um relatório de empréstimos por: membro, código de barras ou intervalo de datas.
Gerar relatório de livros	Operador	Gera um relatório de livros por: título, autor ou coleção.
Gerar relatório de membros	Operador	Gera um relatório de membros por: nome, cidade ou categoria.
Devolver	Operador	Registra a devolução de um item (exemplar).
Emprestar	Operador	Registra o empréstimo de um item (exemplar).
Registrar novos livros	Bibliotecária	Registra novos livros no catálogo da biblioteca.
Criar exemplares	Bibliotecária	Adiciona um novo item (exemplar) com código de barras em um livro existente.
Criar autores	Bibliotecária	Cria novos autores para utilizar dentro dos livros.
Criar editoras	Bibliotecária	Cria novas editoras para usar dentro dos livros.
Criar assuntos	Bibliotecária	Cria novos assuntos para usar dentro dos livros.
Definir a classificação do livro	Bibliotecária	Define classificações de livros (música, medicina, ciência, etc.).
Definir estados de exemplares	Bibliotecária	Define estado dos itens (disponível, emprestado, perdido, etc.).
Definir coleções de livros	Bibliotecária	Define coleções de materiais (livro, periódico, dvd, etc.).
Criar usuário do sistema	Administrador	Cria novos usuários do sistema.
Definir papel do usuário	Administrador	Define o papel do usuário do sistema (bibliotecária ou operador).

7.1.6 Logins e perfis

Conforme pode ser visto na figura a seguir, após realizar o download, e proceder com a instalação do sistema Library, você irá se deparar com a tela de login do sistema, e precisará de um usuário e senha para acessar o sistema com algum de seus perfis.

Figura 175 Tela de entrada da aplicação Library

Para acessar o sistema corretamente, preparamos para você uma tabela com os logins que podem ser utilizados. Nesta tabela, temos três logins, sendo um para cada papel de usuário no sistema.

Tabela 12. Logins e senhas para a aplicação Library

Usuário	Login	Senha	Papel
Administrator	admin	admin	Administrador
Ana Librarian	ana	test	Bibliotecária
Luciele Operator	luciele	test	Operador

7.2 Aplicação Changeman

A segunda aplicação que vamos estudar se chama Changeman, em referência a “Change Manager”. O Changeman tem como objetivo gerenciar solicitações de mudanças em software. Muitas vezes, software deste tipo também são chamados de *bug trackers*, *issue trackers*, ou simplesmente, sistema de chamados.

O software Changeman é focado em mudanças de software, logo sua modelagem reflete este conceito. O Changeman é um software voltado à diferentes perfis de usuários: cliente, membro da equipe (analista, desenvolvedor), gerente de projeto e administrador. Da mesma forma que no software Library, no Changeman as opções do software também se moldam conforme o perfil do usuário que está logado.

O Changeman possui recursos como: abertura e fechamento de tickets com alertas via e-mail aos usuários e técnicos; anotações em tickets; consulta ao histórico de tickets, dentre outros. O Changeman é uma aplicação relativamente simples, mas completamente funcional, frente às soluções comerciais mais completas.

Obs: O Changeman pode ser baixado em <http://www.adianti.com.br/framework-changeman>.

7.2.1 Conteúdo da aplicação

O Changeman é construído sobre a estrutura do Template, e fornece o seguinte conteúdo específico e adicional sobre esta estrutura:

Tabela 13. Conteúdos do Changeman

app	Diretório da aplicação
config changeman.ini	Pasta com configurações. Configuração de acesso para a base de chamados.
control changeman	Classes de controle. Classes de controle de chamados.
database changeman.db changeman.sql	Arquivos de base de dados em formato SQLite. Base de dados em SQLite para chamados. Script de criação da base de chamados.
model changeman	Classes de modelo (Active Records). Classes de modelo de chamados.

7.2.2 Diagrama de classes

Na figura a seguir, podemos conferir o diagrama de classes do sistema, com as classes da camada de modelo do sistema (`app/model`). Essas entidades são responsáveis por gerenciar os dados, bem como as regras de negócios do sistema.

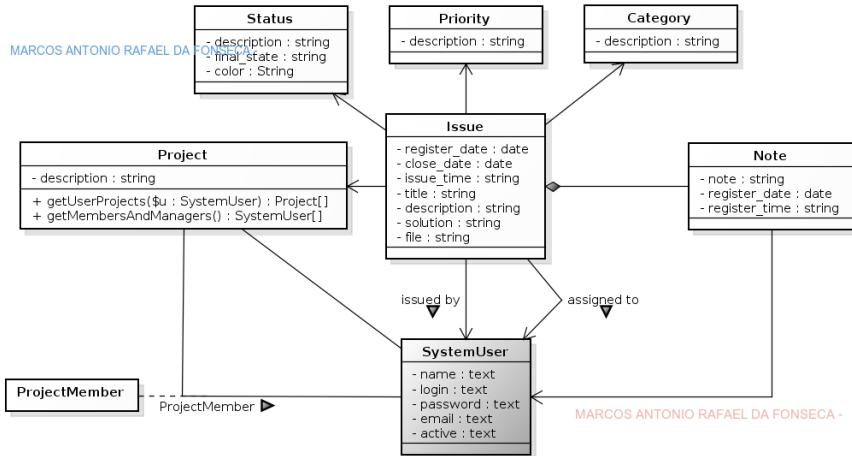


Figura 176 Diagrama de classes do changeman

A seguir, uma lista com a descrição das classes de modelo:

- Priority**: a prioridade de um ticket (Ex: Urgente, desejável);
- Category**: a categoria de um ticket (Ex: Correção de bug);
- Status**: o estado de um ticket (Ex: aberto, em andamento);
- Issue**: representa um ticket (chamado técnico). Possui associações com **Status**, **Priority**, **Category**, e **Project**. Possui uma composição com **Note**, para indicar as anotações realizadas dentro do ticket;
- Note**: representa uma anotação realizada em um ticket;
- Project**: representa um projeto.

7.2.3 Modelo relacional

O modelo relacional do Changeman, é composto pelas seguintes tabelas:

- priority**: armazena um Active Record do tipo **Priority**;
- category**: armazena um Active Record do tipo **Category**;
- status**: armazena um Active Record do tipo **Status**;
- issue**: armazena um Active Record do tipo **Issue**. As associações de **Issue** com as classes: **Status**, **Priority**, **Category**, **Project** e **Member** (**issued by**) e **Member** (**assigned to**), foram mapeadas por meio das chaves estrangeiras **id_status**, **id_priority**, **id_category**, **id_project**, **id_member** e **id_user**;
- note**: armazena um Active Record do tipo **Note**. A composição de **Issue** com **Note** foi mapeada por meio da chave estrangeira **id_issue**. A associação com **Member**, por meio da chave estrangeira **id_user**;
- project**: armazena um Active Record do tipo **Project**.

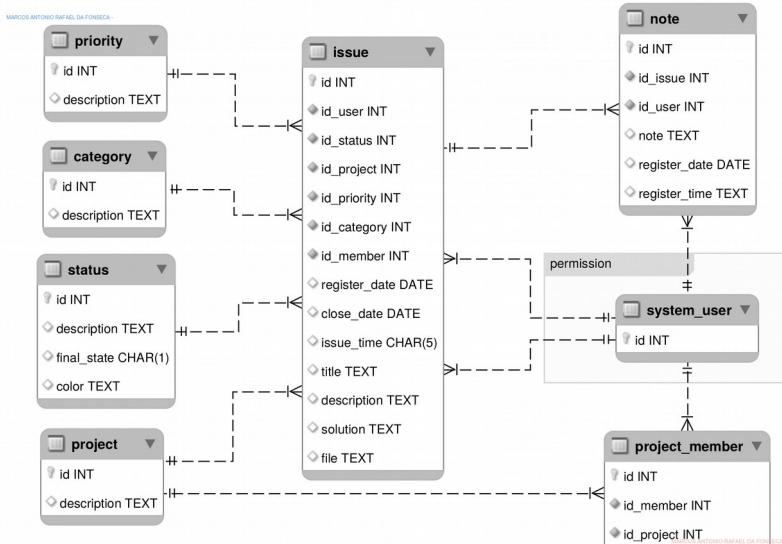


Figura 177 Modelo relacional do changeman

7.2.4 Diagrama de casos de uso

O diagrama de casos de uso a seguir, procura demonstrar os principais papéis de usuários no sistema, bem como as suas responsabilidades. O papel mais comum é o usuário, que representa o cliente. Membro da equipe representa um colaborador da equipe (analista, desenvolvedor). O que o usuário pode fazer, membro da equipe também pode. Gerente representa um gerente de equipe de projeto. O que um membro da equipe pode fazer, também um gerente pode. Por fim, temos o administrador, que herda todos os casos de uso dos demais atores, e ainda, pode realizar uma série de operações específicas, como a criação de novos projetos.

O papel “Usuário” representa o cliente do sistema, que pode realizar abertura de tickets, adicionar notas, consultar o seu histórico de tickets abertos, dentre outros. Já o “Membro da equipe”, representa um técnico (desenvolvedor, analista), e pode, além disso, editar tickets e registrar seu fechamento. O gerente por sua vez, pode excluir tickets e cadastrar estados, prioridades e categorias de tickets, além de cadastrar projetos, e vincular a equipe (membros) dentro de cada projeto. O administrador do sistema pode criar usuários, grupos, e definir as permissões básicas.

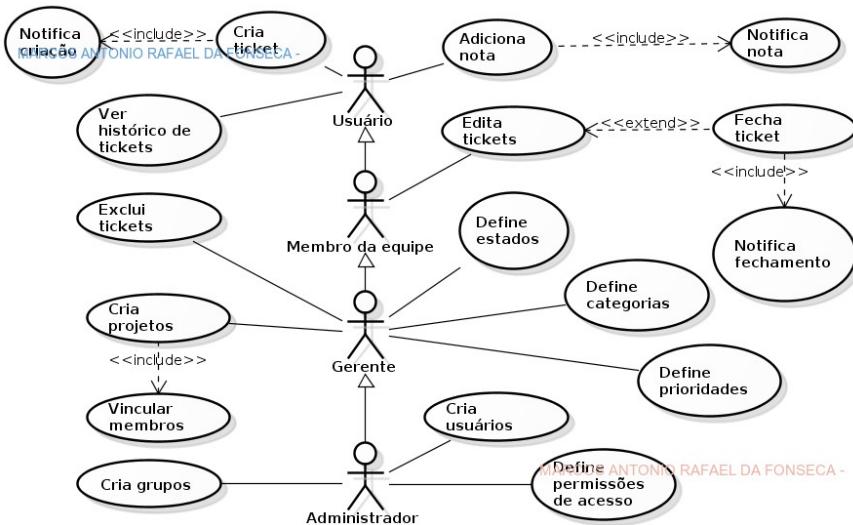


Figura 178 Diagrama de casos de uso do Changeman

7.2.5 Especificação dos casos de uso

A tabela a seguir, procura explicar de maneira mais completa as funcionalidades que cada papel poderá acessar dentro do sistema.

Tabela 14. Papéis e responsabilidades para o sistema Changeman

Caso de uso	Papel	Descrição
Cria um ticket	Usuário	Preenche um novo ticket com: projeto, prioridade, categoria, título, descrição.
Notifica criação	Sistema	Quando alguém cria um novo ticket, o sistema notifica os membros e gerentes do projeto.
Adiciona nota	Usuário	O usuário pode acrescentar uma nota textual ao ticket.
Notifica nota	Sistema	Quando alguém adicionar uma nota, o sistema notifica os membros e gerentes do projeto.
Ver histórico de tickets	Usuário	Permite o usuário ver os tickets já preenchidos.
Editar ticket	Membro da equipe	Permite que membros da equipe (desenvolvedores, analistas) editem um ticket, alterando as suas propriedades.
Fecha ticket	Membro da equipe	Quando um membro da equipe altera o status para fechado, e preenche a solução, a solicitação é considerada fechada. Neste caso, o sistema notifica o usuário, membros da equipe e gerentes (manager).
Notifica fechamento	Sistema	O sistema notifica o usuário, membros da equipe e gerentes sobre o fechamento de um ticket.
Excluir ticket	Gerente	O gerente pode excluir tickets.
Cria projetos	Gerente	O gerente pode criar projetos, e vincular membros ao projeto.
Define estados	Gerente	O gerente pode criar status de ticket (new, closed, rejected).
Define categorias	Gerente	O gerente pode criar categorias de ticket (bug, new feature, task).
Define prioridades	Gerente	O gerente pode criar prioridades de ticket (low, normal, high).
Cria grupos	Administrador	O administrador cria grupos de usuários.
Cria usuários	Administrador	O administrador cria usuários.
Define permissões de acesso	Administrador	O administrador define permissões de acesso.

7.2.6 Logins e perfis

Conforme pode ser visto na figura a seguir, após realizar o download e instalação do sistema Changeman, você irá se deparar com a tela de login do sistema, e precisará de um usuário e senha para acessar o sistema com algum de seus perfis.



Figura 179 Tela de entrada do sistema Changeman

Para acessar o sistema corretamente, preparamos para você uma tabela com os logins que podem ser utilizados. Nesta tabela, temos quatro logins, sendo um para cada papel de usuário no sistema.

Tabela 15. Logins e senhas para a aplicação Changeman

Usuário	Login	Senha	Papel
Administrador	admin	admin	Administrador
John Manager	john	test	Gerente
Daniel Developer	daniel	test	Membro da equipe
Mary Customer	mary	test	Usuário

CAPÍTULO 8

Interações

O objetivo deste capítulo é abordar alguns tópicos adicionais relacionados a integração de outras bibliotecas dentro de nossa aplicação, bem como da integração de nossa aplicação com outras por meio de Web Services.

8.1 Rotas amigáveis

Ao criarmos uma aplicação com o Framework, percebemos que a URL nativa contém a informação da classe e do método que será executado, no seguinte formato:

```
http://www.aplicacao.local/index.php?class=ContactForm&method=onEdit&key=1
```

Como o Framework é utilizado na grande maioria das vezes para criar sistemas corporativos internos, a URL não é tão relevante quanto seria em um web site público ou e-commerce. Mas mesmo em contextos privados e corporativos, é interessante usarmos URL's em um formato amigável para facilitar o compartilhamento de links entre colaboradores. Além disso, o Framework é cada vez mais usado para produzir diferentes tipos de aplicações, como e-commerce, portais corporativos, etc. Dessa forma, vamos verificar como transformar uma URL completa em uma URL resumida:

```
http://www.aplicacao.local/contact-edit?key=1
```

Aplicação de exemplo

Uma aplicação com rotas pré-configuradas é disponibilizada para download no site da Adianti. Ela se chama Contacts API, e além de rotas amigáveis possui outros recursos como serviços REST e RESTful. A aplicação Contacts API já possui rotas pré-definidas para todas as ações previstas no Template, tais como: gerenciamento de programas, grupos de usuários, unidades, preferências, documentos, dentre outros. Caso tenha dúvidas, faça download da Contacts API para verificar uma aplicação já em funcionamento com o recurso de rotas amigáveis.

Pré requisitos

Para a abordagem a seguir funcionar, utilizaremos o servidor de páginas Apache. Antes, é preciso configurar o apache para habilitar a sobreescrita de configuração (`AllowOverride`), o que geralmente é feito no `apache2.conf`.

`apache2.conf`

```
<Directory /var/www/>
    Options Indexes FollowSymLinks
    AllowOverride All
</Directory>
```

Configurando as rotas de entrada

O primeiro passo é configurar as rotas no servidor de páginas (Apache, Nginx). Neste exemplo, vamos utilizar o Apache. O primeiro passo é criar um arquivo no diretório raiz da aplicação (`.htaccess`) com as rotas e os seus respectivos direcionamentos. Neste exemplo, estamos criando rotas para o formulário de edição (`contact-edit`), para listagem (`contact-list`), e exclusão de registros (`contact-ondelete`, `contact-delete`), todas apontando para o `index.php`. Caso o usuário digite na URL estes caminhos, o servidor de páginas redirecionará para o `index.php` com os respectivos parâmetros. Além das rotas para o `index.php`, o Framework precisa das rotas para o `engine.php`. Estas rotas iniciam com `xhr-` e são usadas internamente para a navegação no sistema, sem necessidade de recarregamento de toda a página. A navegação interna (post, edição, paginação, ordenação) utiliza as rotas `xhr-` para maior performance.

`.htaccess`

```
RewriteEngine On
RewriteCond %{REQUEST_FILENAME} !-f
RewriteCond %{REQUEST_FILENAME} !-d

#Application specific routes
RewriteRule ^contact-edit$ index.php?class=ContactForm&method=onEdit&%{QUERY_STRING} [NC]
RewriteRule ^contact-list$ index.php?class=ContactList&method=onReload&%{QUERY_STRING} [NC]
RewriteRule ^contact-ondelete$ index.php?class=ContactList&method=onDelete&%{QUERY_STRING} [NC]
RewriteRule ^contact-delete$ index.php?class=ContactList&method=Delete&%{QUERY_STRING} [NC]

RewriteRule ^xhr-contact-edit$ engine.php?class=ContactForm&method=onEdit&%{QUERY_STRING} [NC]
RewriteRule ^xhr-contact-list$ engine.php?class=ContactList&method=onReload&%{QUERY_STRING} [NC]
RewriteRule ^xhr-contact-ondelete$ engine.php?class=ContactList&method=onDelete&%{QUERY_STRING} [NC]
RewriteRule ^xhr-contact-delete$ engine.php?class=ContactList&method=Delete&%{QUERY_STRING} [NC]
```

Obs: O Template já acompanha um arquivo `htaccess-dist`, com rotas padrão. Renomeie-o para `.htaccess` e adicione as suas rotas, conforme sugerido.

Traduzindo as rotas geradas pelo Framework

Até este ponto, configuramos as rotas de entrada na aplicação, o que significa que se o usuário entrar na URL com a rota curta, o sistema já direcionará para o caminho correto. Entretanto, os links gerados internamente pelo Framework devem ser traduzidos para o formato curto também. Links são gerados em diversas situações como em postagens de formulários, ordenação, paginação de datagrids, edição de

datagrids, dentre outros. Nestes casos, o Framework precisa entender que é necessário gerar a rota encurtada no lugar da rota nativa. Para que isso seja possível, é necessário alterar os arquivos `index.php` e `engine.php` para habilitar um tradutor de rotas. O tradutor indicará ao Framework traduzir as rotas extensas em rotas curtas.

index.php

```
AdantiCoreApplication::setRouter(array('AdantiRouteTranslator', 'translate'));
```

engine.php

```
AdantiCoreApplication::setRouter(array('AdantiRouteTranslator', 'translate'));
```

A classe `AdantiRouteTranslator` é responsável por traduzir as rotas geradas pelo Framework para o formato curto. Ela pode funcionar no modo manual ou automático. No modo manual (procure por “manual entries” no arquivo a seguir), você deve criar um vetor com o mapeamento da URL a ser traduzida e sua versão amigável. Já no modo automático, basta executar o método `parseHtAccess()` para realizar a leitura do arquivo `.htaccess` e gerar o mapeamento automaticamente. Assim, se você seguir as instruções e usar Apache com `.htaccess`, não será necessário alterar este arquivo.

app/lib/util/AdantiRouteTranslator.php

```
<?php
class AdantiRouteTranslator
{
    public static function translate($url, $format = TRUE)
    {
        /** manual entries
         * $routes = array();
         * $routes['class=TipoProdutoList'] = 'tipo-produto-list';
         * $routes['class=TipoProdutoList&method=onReload'] = 'tipo-produto-list';
         * $routes['class=TipoProdutoForm&method=onEdit'] = 'tipo-produto-edit';
         * $routes['class=TipoProdutoForm&method=onDelete'] = 'tipo-produto-ondelete';
         * $routes['class=TipoProdutoForm&method=delete'] = 'tipo-produto-delete';
         */
        // automatic parse .htaccess
        $routes = self::parseHtAccess();
        // ...
    }
}
```

Testando

Agora, você já pode navegar na aplicação pelas rotas cadastradas e verá o funcionamento, como no caso a seguir:

<code>http://www.aplicacao.local/contact-edit?key=1</code>
--

Possíveis erros

Caso durante a navegação ocorram erros como “Connection Failed”, é provável que a rota está cadastrada da maneira errada no `.htaccess`, ou o Apache não está com o `AllowOverride` habilitado corretamente.

8.2 Serviços REST

Durante o desenvolvimento de um sistema, é comum termos de disponibilizar algumas de suas rotinas para que estas sejam executadas por aplicativos externos. O caminho mais utilizado é por meio de Web Services, e REST é um dos meios para se obter isto. Imagine que você desenvolveu um sistema, e agora precisa que outros sistemas possam consultar informações a respeito de seus objetos. Nesta seção, veremos como disponibilizar o acesso a objetos por meio de Web Services. Para tal, trabalharemos com a classe de Contatos, e criaremos um Web Service para manipulá-los.

Aplicação de exemplo

Uma aplicação com exemplos de serviços REST é disponibilizada para download no site da Adianti. Ela se chama Contacts API, e inclui serviços REST e RESTful.

Habilitando o servidor

O primeiro passo é habilitar um servidor REST no Adianti Framework. Para tal, basta renomearmos o arquivo `rest.php.dist` para `rest.php` e o servidor estará no ar. Este arquivo faz parte da distribuição padrão do Template, e está no diretório principal.

Como este arquivo disponibiliza as classes do Framework para utilização externa, é imprescindível implementar um mecanismo de controle, ou seja, filtrar somente as requisições válidas. Para tal, foi colocado ali no arquivo um lembrete (*// aqui implementar mecanismo de controle*).

Classe de modelo

Antes de disponibilizar um serviço, é necessário termos a classe de entidade previamente declarada na pasta `app/model` ou em algum subdiretório. Se você já usa o Framework, provavelmente já tem várias classes de modelo.

`app/model/Contact.php`

```
<?php
class Contact extends TRecord
{
    const TABLENAME = 'contact';
    const PRIMARYKEY= 'id';
    const IDPOLICY = 'max'; // {max, serial}

    public function __construct($id = NULL, $callObjectLoad = TRUE)
    {
        parent::__construct($id, $callObjectLoad);
        parent::addAttribute('name');
        parent::addAttribute('email');
        parent::addAttribute('number');
        parent::addAttribute('address');
        parent::addAttribute('notes');
    }
}
```

Criando a classe de serviço

Provavelmente você já tem uma série de classes de modelo representando suas tabelas na pasta `app/model`. Agora, queremos criar um Web Service para disponibilizar uma API para manipular uma classe de modelo. Para tal, precisamos criar uma classe de serviço. Uma classe de serviço cria uma camada que interage com uma funcionalidade interna da aplicação e a expõe para o mundo externo. Neste caso, vamos criar a classe `app/service/ContactService.php`. É necessário configurar ainda a constante `DATABASE` com o nome do conector de base de dados (existente em `app/config/contacts.ini`), e a constante `ACTIVE_RECORD` para conter o nome da classe de modelo (existente em `app/model/Contact.php`) que será disponibilizada. Ao usarmos a classe `AdiantiRecordService` como base (herança), a classe de serviço já herdará uma série de métodos REST como: `load()`, `store()`, `delete()`, `loadAll()`, e `deleteAll()`.

app/service/ContactRestService.php

```
<?php
class ContactRestService extends AdiantiRecordService
{
    const DATABASE      = 'contacts';
    const ACTIVE_RECORD = 'Contact';
}
```

Preparando os testes

Para preparar os testes, precisaremos de uma função para realizar POST's para uma URL do sistema. Para tal, vamos criar uma função chamada `request()`, que receberá a localização do serviço no primeiro parâmetro, o método de envio (POST, PUT, GET, DELETE) no segundo parâmetro, e um vetor com os dados da postagem no terceiro parâmetro. Esta função de testes acompanha a aplicação de exemplo Contacts API.

Carregar registro

Para carregar um registro, realizamos um request do tipo GET para o REST Server (`rest.php`), identificando a classe de serviço (`ContactRestService`), o método (`load`), e o `id` do registro a ser carregado.

rest/rest_load.php

```
<?php
require_once 'request.php';

try
{
    $location = 'http://localhost/contacts/rest.php';

    $parameters = [];
    $parameters['class']  = 'ContactRestService';
    $parameters['method'] = 'load';
    $parameters['id']     = '1';

    print_r(request($location, 'GET', $parameters));
}
```

```

catch (Exception $e)
{
    echo 'Error: '. $e->getMessage();
}

catch (Exception $e)
{
    echo 'Error: '. $e->getMessage();
}

```

O registro será retornado na forma de objeto.

Retorno:

```

stdClass Object
(
    [id] => 1
    [name] => Mary
    [email] => mary@mary.com
    [number] => +55 (51) 1234-5678
    [address] => Mary Street, 123, Philadelphia
    [notes] => Good customer
)

```

Criar registro

Para criar um registro, realizamos um request do tipo `POST` para o REST Server (`rest.php`), identificando a classe de serviço (`ContactRestService`), o método (`store`), e o vetor com os dados a serem gravados (`data`). Cada posição do vetor de dados conterá um atributo a ser armazenado na tabela. Como não estamos informando o `id`, a classe realizará um `insert`.

rest/rest_new.php

```

<?php
require_once 'request.php';

try
{
    $location = 'http://localhost/contacts/rest.php';

    $parameters = [];
    $parameters['class'] = 'ContactRestService';
    $parameters['method'] = 'store';
    $parameters['data'] = ['name' => 'Peter',
                          'email' => 'peter@email.com',
                          'number' => '12345678',
                          'address' => 'Peter St, 123'];

    print_r(request($location, 'POST', $parameters));
}
catch (Exception $e)
{
    echo 'Error: '. $e->getMessage();
}

```

O registro será retornado na forma de objeto, com o `id` gerado.

Retorno

```
stdClass Object
(
    [name] => Peter
    [email] => peter@email.com
    [number] => 12345678
    [address] => Peter St, 123
    [id] => 4
)
```

Alterar registro

Para alterar um registro, realizarmos um request do tipo **POST** para o REST Server (**rest.php**), identificando a classe de serviço (**ContactRestService**), o método (**store**), e o vetor com os dados a serem gravados (**data**), a alteração ocorre pois o **id** é informado. Caso contrário, é criado um novo registro.

rest/rest_update.php

```
<?php
require_once 'request.php';

try
{
    $location = 'http://localhost/contacts/rest.php';

    $parameters = [];
    $parameters['class'] = 'ContactRestService';
    $parameters['method'] = 'store';
    $parameters['data'] = ['name' => 'Dino Sauro', 'id' => 3];

    print_r(request($location, 'POST', $parameters));
}
catch (Exception $e)
{
    echo 'Error: ' . $e->getMessage();
}
```

O retorno terá um objeto com os dados informados.

Retorno

```
stdClass Object
(
    [name] => Dino Sauro
    [id] => 3
)
```

Excluir registro

Para excluir um registro, realizarmos um request do tipo **POST** para o REST Server (**rest.php**), identificando a classe de serviço (**ContactRestService**), o método (**delete**), e o **id** a ser excluído.

rest/rest_delete.php

```
<?php
require_once 'request.php';

try
{
    $location = 'http://localhost/contacts/rest.php';

    $parameters = [];
    $parameters['class'] = 'ContactRestService';
    $parameters['method'] = 'delete';
    $parameters['id'] = 3;

    request($location, 'POST', $parameters);
}
catch (Exception $e)
{
    echo 'Error: ' . $e->getMessage();
}
```

Carregar vários registros

Para listar todos os registros, realizarmos um request do tipo GET para o REST Server (`rest.php`), identificando a classe de serviço (`ContactRestService`), o método (`loadAll`), e os possíveis filtros da consulta na forma de array (`filters`).

rest/rest_all.php

```
<?php
require_once 'request.php';

try
{
    $location = 'http://localhost/contacts/rest.php';

    $parameters = [];
    $parameters['class'] = 'ContactRestService';
    $parameters['method'] = 'loadAll';
    // $parameters['filters'] = [ ['id', '>', 1 ] ];

    print_r(request($location, 'GET', $parameters));
}
catch (Exception $e)
{
    echo 'Error: ' . $e->getMessage();
}
```

O retorno será um vetor de objetos.

Retorno

```
Array
(
    [0] => stdClass Object
        (
            [name] => Mary
            [email] => mary@mary.com
            [number] => +55 (51) 1234-5678
            [address] => Mary Street, 123, Philadelphia
            [notes] => Good customer
            [id] => 1
        )
)
```

```
[1] => stdClass Object
(
    [name] => John
    [email] => john@mail.com
    [number] => +55 (51) 1234-5678
    [address] => Jon Street, 123, Philadelphia
    [notes] => Good customer
    [id] => 2
)
...
...
```

Excluir vários registros

Para excluir vários registros, realizarmos um request do tipo POST para o REST Server (`rest.php`), identificando a classe de serviço (`ContactRestService`), o método (`deleteAll`), e os filtros com os registros a serem excluídos (`filters`).

rest/rest_deleteall.php

```
<?php
require_once 'request.php';

try
{
    $location = 'http://localhost/contacts/rest.php';

    $parameters = [];
    $parameters['class'] = 'ContactRestService';
    $parameters['method'] = 'deleteAll';
    $parameters['filters'] = [ ['id', '>', 3] ];

    print_r(request($location, 'POST', $parameters));
}
catch (Exception $e)
{
    echo 'Error: ' . $e->getMessage();
}
```

Executando um método próprio

Nem sempre os métodos básicos (`load`, `store`, `delete`) são suficientes. Existem situações, onde é necessário disponibilizar um método personalizado com alguma regra de negócio diferenciada. Nestes casos, podemos criar e disponibilizar um método próprio da classe de serviço. Neste caso, vamos criar um método chamado `getBetween()` que receberá dois parâmetros: `from` e `to`, e deverá retornar todos os registros entre esses dois códigos.

app/service/ContactRestService.php

```
<?php
class ContactRestService extends AdiantiRecordService
{
    const DATABASE      = 'contacts';
    const ACTIVE_RECORD = 'Contact';
```

```
public static function getBetween( $request )
{
    TTransaction::open('contacts');
    $response = array();

    // carrega os contatos
    $all = Contact::where('id', '>=', $request['from'])
        ->where('id', '<=', $request['to'])
        ->load();
    foreach ($all as $product)
    {
        $response[] = $product->toArray();
    }
    TTransaction::close();
    return $response;
}
```

No exemplo a seguir, veremos como consumir o método próprio. Para tal, realizarmos um request (**POST**) para o REST Server (**rest.php**), identificando a classe de serviço (**ContactRestService**), o método (**getBetween**), e os parâmetros (**from**, **to**) necessários. O retorno será na forma de um Array de objetos.

rest/rest_method.php

```
<?php
require_once 'request.php';

try
{
    $location = 'http://localhost/contacts/rest.php';
    $parameters = array();
    $parameters['class']      = 'ContactRestService';
    $parameters['method']     = 'getBetween';
    $parameters['from']       = '1';
    $parameters['to']         = '3';

    print_r(request($location, 'GET', $parameters));
}
catch (Exception $e)
{
    echo 'Error: ' . $e->getMessage();
}
```

8.3 Serviços RESTful

Na seção anterior, vimos como criar um serviço REST. Agora vamos proceder com a criação de um serviço RESTful.

Aplicação de exemplo

Uma aplicação com exemplos de serviços REST é disponibilizada para download no site da Adianti. Ela se chama Contacts API, e inclui serviços REST e RESTful.

Habilitando o servidor

O primeiro passo é habilitar um servidor REST no Adianti Framework. Para tal, renomear o arquivo `rest.php.dist` para `rest.php` que consta no diretório principal do Template e servidor estará no ar.

Como este arquivo disponibiliza as classes do Framework para utilização externa, é imprescindível implementar um mecanismo de controle, ou seja, filtrar somente as requisições válidas. Para tal, foi colocado ali no arquivo um lembrete (*// aqui implementar mecanismo de controle*). Na sequência, abordaremos segurança em serviços REST.

Classe de modelo

Antes de disponibilizar um serviço, é necessário termos a classe de entidade previamente declarada na pasta `app/model` ou em algum subdiretório. Se você já usa o Framework, provavelmente já tem várias classes de modelo.

`app/model/Contact.php`

```
<?php
class Contact extends TRecord
{
    const TABLENAME = 'contact';
    const PRIMARYKEY= 'id';
    const IDPOLICY = 'max'; // {max, serial}

    public function __construct($id = NULL, $callObjectLoad = TRUE)
    {
        parent::__construct($id, $callObjectLoad);
        parent::addAttribute('name');
        parent::addAttribute('email');
        parent::addAttribute('number');
        parent::addAttribute('address');
        parent::addAttribute('notes');
    }
}
```

Criando a classe de serviço

Provavelmente você já tem uma série de classes de modelo representando suas tabelas na pasta `app/model`. Agora, queremos criar um Web Service para disponibilizar uma API para manipular uma classe de modelo. Para tal, precisamos criar uma classe de serviço. Uma classe de serviço cria uma camada que interage com uma funcionalidade interna da aplicação e a expõe para o mundo externo. Neste caso, vamos criar a classe

`app/service/ContactService.php`. É necessário configurar ainda a constante `DATABASE` com o nome do conector de base de dados (existente em `app/config/contacts.ini`), e a constante `ACTIVE_RECORD` para conter o nome da classe de modelo (existente em `app/model/Contact.php`) que será disponibilizada. Ao usarmos a classe `AdiantiRecordService` como base (herança), a classe de serviço já herdará uma série de métodos REST como: `load()`, `store()`, `delete()`, `loadAll()`, e `deleteAll()`.

`app/service/ContactRestService.php`

```
<?php
class ContactRestService extends AdiantiRecordService
{
    const DATABASE      = 'contacts';
    const ACTIVE_RECORD = 'Contact';
}
```

Configurando as rotas

Para cada classe de serviço a ser disponibilizada, devem ser criadas três rotas. A primeira é usada para operações com objetos individuais como um GET ou PUT para `/contacts/1`, já a segunda é usada para ações personalizadas sobre objetos individuais como `/contacts/1/action`, e a terceira é usada para operações em lote como um GET ou POST para `/contacts`. Para cada nova classe de serviço criada, você deve replicar essas três linhas, substituindo “`contacts`” do primeiro segmento pelo nome do objeto manipulado, e “`ContactRestService`” do segundo segmento pelo nome da classe de serviço a ser disponibilizada.

`.htaccess`

```
#RESTFUL routes
RewriteRule ^contacts/([A-Za-z0-9]*)$ rest.php?class=ContactRestService&method=handle&id=$1&%{QUERY_STRING} [NC]
RewriteRule ^contacts/([A-Za-z-_0-9]*)/([A-Za-z-_0-9]*)$ rest.php?
class=ContactRestService&method=$2&id=$1&%{QUERY_STRING} [NC]
RewriteRule ^contacts$ rest.php?class=ContactRestService&method=handle&%{QUERY_STRING} [NC]
```

Carregar registro em PHP

Para carregar um registro em PHP, realizamos um request do tipo GET para a URL contendo o endpoint com o registro a ser consultado `/contacts/<id>`.

`rest/restful_load.php`

```
<?php
require_once 'request.php';

try
{
    $location = 'http://localhost/sistema/contacts/1';
    print_r( request($location, 'GET') );
}
catch (Exception $e)
{
    echo 'Error: ' . $e->getMessage();
}
```

O retorno será um objeto com os dados.

Retorno

```
stdClass Object
(
    [id] => 1
    [name] => Mary
    [email] => mary@mary.com
    [number] => +55 (51) 1234-5678
    [address] => Mary Street, 123 Philadelphia
    [notes] => Good customer
)
```

Carregar registro em Javascript

Para carregar um registro em Javascript, realizamos um request do tipo GET para a URL contendo o endpoint com o registro a ser consultado `/contacts/<id>`.

```
$ajax({
    type: 'GET',
    url: 'http://localhost/sistema/contacts/1',
    success: function (response) {
        console.log(response.data);
    }
});
```

O retorno será um objeto com os dados.

Retorno

```
{
    "id": "1",
    "name": "Mary",
    "email": "mary@mary.com",
    "number": "+55 (51) 1234-5678",
    "address": "Mary Street, 123 Philadelphia",
    "notes": "Good customer"
}
```

Carregar registro em Shell

Para carregar um registro em Shell, realizamos um request do tipo GET para a URL contendo o endpoint com o registro a ser consultado `/contacts/<id>`.

```
curl -i -X GET http://localhost/sistema/contacts/1
```

O retorno será um objeto com os dados.

Retorno

```
HTTP/1.1 200 OK
{"status": "success", "data": {"id": "1", "name": "Mary", "email": "mary@mary.com", "number": "+55 (51) 1234-5678", "address": "Mary Street, 123 Philadelphia", "notes": "Good customer"}}
```

Criar registro em PHP

Para criar um registro em PHP, realizamos um request do tipo **POST** para a URL contendo o endpoint **/contacts**, identificando no terceiro parâmetro o vetor com os dados do objeto.

rest/restful_new.php

```
<?php
require_once 'request.php';

try
{
    $body = [];
    $body['name'] = 'Dino';
    $location = 'http://localhost/sistema/contacts';
    print_r( request($location, 'POST', $body) );
}
catch (Exception $e)
{
    echo 'Error: ' . $e->getMessage();
}
```

O retorno será um objeto com os dados.

Retorno

```
stdClass Object
(
    [name] => Dino
    [id] => 3
)
```

Criar registro em Javascript

Para criar um registro em Javascript, realizamos um request do tipo **POST** para a URL contendo o endpoint **/contacts**, identificando os dados do objeto na posição data.

```
$.ajax({
    type: 'POST',
    url: 'http://localhost/sistema/contacts',
    data: {
        'name': 'Dino',
        'email': 'dino@mail.com'
    },
    dataType: 'json',
    success: function (response) {
        console.log(response.data);
    }
});
```

O retorno será um objeto com os dados.

Retorno

```
{
    "name": "Dino",
    "email": "dino@mail.com",
    "id": 4
}
```

Criar registro em Shell

Para criar um registro em Shell, realizamos um request do tipo **POST** para a URL contendo o endpoint **/contacts**, identificando os dados do objeto no parâmetro “**-d**”.

```
curl -i -X POST -H 'Content-Type: application/json' -d '{"name": "Dino", "email" : "dino@email.com"}' http://localhost/sistema/contacts
```

O retorno será um objeto com os dados.

Retorno

```
HTTP/1.1 200 OK
{"status": "success", "data": {"name": "Dino", "email": "dino@email.com", "id": 5}}
```

Alterar registro em PHP

Para alterar um registro em PHP, realizamos um request do tipo **PUT** para a URL contendo o endpoint **/contacts/<id>**, identificando os dados para alteração do objeto no terceiro parâmetro.

rest/restful_update.php

```
<?php
require_once 'request.php';

try
{
    $body = [];
    $body['name']      = 'Dino Sauro';
    $location = 'http://localhost/sistema/contacts/3';
    print_r( request($location, 'PUT', $body) );
}
catch (Exception $e)
{
    echo 'Error: ' . $e->getMessage();
}
```

O retorno será um objeto com os dados.

Retorno

```
stdClass Object
(
    [id] => 3
    [name] => Dino Sauro
)
```

Alterar registro em Javascript

Para alterar um registro em Javascript, realizamos um request do tipo **PUT** para a URL contendo o endpoint **/contacts/<id>**, identificando os dados para alteração do objeto no parâmetro **data**.

```
$.ajax({
    type: 'PUT',
    contentType: 'application/json',
    url: 'http://localhost/sistema/contacts/3',
    data: JSON.stringify({
        'name': 'Dino Sauro',
    }),
    dataType: 'json',
    success: function (response) {
        console.log(response.data);
    }
});
```

O retorno será um objeto com os dados.

Retorno

```
{
    "id": "3",
    "name": "Dino Sauro"
}
```

Alterar registro em Shell

Para alterar um registro em Shell, realizamos um request do tipo `PUT` para a URL contendo o endpoint `/contacts/<id>`, identificando os dados para alteração do objeto no parâmetro “-d”.

```
curl -i -X PUT -H 'Content-Type: application/json' -d '{"name": "Dino Sauro"}'
http://localhost/sistema/contacts/3
```

O retorno será um objeto com os dados.

Retorno

```
HTTP/1.1 200 OK
{"status": "success", "data": {"id": "3", "name": "Dino Sauro"}}
```

Excluir registro em PHP

Para excluir um registro em PHP, realizamos um request do tipo `DELETE` para a URL contendo o endpoint `/contacts/<id>`.

rest/restful_delete.php

```
<?php
require_once 'request.php';

try
{
    $location = 'http://localhost/sistema/contacts/3';
    print_r( request($location, 'DELETE') );
}
catch (Exception $e)
{
    echo 'Error: ' . $e->getMessage();
}
```

Excluir registro em Javascript

Para excluir um registro em Javascript, realizamos um request do tipo **DELETE** para a URL contendo o endpoint `/contacts/<id>`.

```
$.ajax({
    type: 'DELETE',
    contentType: 'application/json',
    url: 'http://localhost/sistema/contacts/3',
    dataType: 'json',
    success: function (response) {
        console.log(response.data);
    }
});
```

Excluir registro em Shell

Para excluir um registro em Shell, realizamos um request do tipo **DELETE** para a URL contendo o endpoint `/contacts/<id>`.

```
curl -i -X DELETE http://localhost/sistema/contacts/3
```

Retorno

```
{"status": "success", "data": null}
```

Carregar vários registros em PHP

Para carregar vários registros em PHP, realizamos um request do tipo **GET** para a URL contendo o endpoint `/contacts`, identificando no terceiro parâmetro ordem, limit e filtros.

rest/restful_all.php

```
<?php
require_once 'request.php';

try
{
    $body['limit'] = '3';
    $body['order'] = 'name';
    $body['direction'] = 'desc';
    // $body['filters'] = [ [ 'id', '>', 1 ] ];

    $location = 'http://localhost/sistema/contacts';
    print_r( request($location, 'GET', $body) );
}
catch (Exception $e)
{
    echo 'Error: ' . $e->getMessage();
}
```

O retorno será um objeto com os dados.

Retorno

```
Array
(
    [0] => stdClass Object
        (
            [name] => Mary
            [email] => mary@mary.com
            [number] => +55 (51) 1234-5678
            [address] => Mary Street, 123, Philadelphia
            [notes] => Good customer
            [id] => 1
        )

    [1] => stdClass Object
        (
            [name] => John
            [email] => john@mail.com
            [number] => +55 (51) 1234-5678
            [address] => Jon Street, 123, Philadelphia
            [notes] => Good customer
            [id] => 2
        )

    ...
)
```

Carregar vários registros em Javascript

Para carregar vários registros em Javascript, realizamos um request do tipo **GET** para a URL contendo o endpoint **/contacts**, identificando parâmetros como ordem, limit e filtros.

```
$.ajax({
    type: 'GET',
    url: 'http://localhost/sistema/contacts',
    data: {
        'order': 'name',
        'direction': 'desc',
        'limit': '3',
        'filters' :[['id",">=",1],["id","<=", "10"]]
    },
    dataType: 'json',
    success: function (response) {
        console.log(response.data);
    }
});
```

O retorno será um objeto com os dados.

Retorno

```
[
    {
        "name": "Mary",
        "email": "mary@mary.com",
        "number": "+55 (51) 1234-5678",
        "address": "Mary Street, 123 Philadelphia",
        "notes": "Good customer",
        "id": "1"
    },
]
```

```
{
    "name": "John",
    "email": "john@mail.com",
    "number": "+55 (51) 1234-5678",
    "address": "Jon Street, 123 Philadelphia",
    "notes": "Good customer",
    "id": "2"
}
]
```

Carregar vários registros em Shell

Para carregar vários registros em Shell, realizamos um request do tipo GET para a URL contendo o endpoint `/contacts`, identificando no argumento “`-d`” parâmetros como ordem, limit e filtros.

```
curl -i -X GET -d '{"order": "name", "direction": "desc", "limit": 3}'
http://localhost/sistema/contacts
```

Retorno

HTTP/1.1 200 OK
`{"status": "success", "data": [{"name": "Mary", "email": "mary@mary.com", "number": "+55 (51) 1234-5678", "address": "Mary Street, 123\r\nPhiladelphia", "notes": "Good customer", "id": "1"}, {"name": "John", "email": "john@mail.com", "number": "+55 (51) 1234-5678", "address": "Jon Street, 123\r\nPhiladelphia", "notes": "Good customer", "id": "2"}]}`

Excluir vários registros em PHP

Para excluir vários registros em PHP, realizamos um request do tipo `DELETE` para a URL contendo o endpoint `/contacts`, identificando no terceiro parâmetro os filtros.

rest/restful_deleteall.php

```
<?php
require_once 'request.php';

try
{
    $location = 'http://localhost/sistema/contacts';
    $body = [];
    $body['filters'] = [ ['id', '>', 1], ['id', '<', '3']];
    print_r( request($location, 'DELETE', $body) );
}
catch (Exception $e)
{
    echo 'Error: ' . $e->getMessage();
}
```

Excluir vários em Javascript

Para excluir vários registros em Javascript, realizamos um request do tipo `DELETE` para a URL contendo o endpoint `/contacts`, identificando no parâmetro data, os filtros.

```
$.ajax({
    type: 'DELETE',
    contentType: 'application/json',
    url: 'http://localhost/sistema/contacts',
    data: JSON.stringify({
        'filters' :[[{"id": ">", 5}, {"id": "<", "10"}]]
    }),
    dataType: 'json',
    success: function (response) {
        console.log(response.data);
    }
});
```

Excluir vários registros em Shell

Para excluir vários registros em Shell, realizamos um request do tipo **DELETE** para a URL contendo o endpoint **/contacts**, identificando no argumento “-d”, os filtros.

```
curl -i -X DELETE -d '{"filters": [{"id": ">", 5}, {"id": "<", "10"}]}'
http://localhost/sistema/contacts
```

Retorno

```
HTTP/1.1 200 OK
{"status": "success", "data": 0}
```

Executando um método próprio

Nem sempre os métodos básicos (**load**, **store**, **delete**) são suficientes. Existem situações, onde é necessário disponibilizar um método personalizado com alguma regra de negócio diferenciada. Nesses casos, podemos criar um método próprio da classe de serviço. Neste caso, vamos criar um método chamado **name()** que retornará o nome do registro solicitado. Claro que isso não é necessário pois o **GET** já carrega o objeto inteiro, mas podemos realizar outras operações mais complexas também.

app/service/ContactRestService.php

```
<?php
class ContactRestService extends AdiantiRecordService
{
    const DATABASE      = 'contacts';
    const ACTIVE_RECORD = 'Contact';

    public function name( $request )
    {
        TTransaction::open('contacts');
        $name = Contact::find($request['id'])->name;
        TTransaction::close();
        return $name;
    }
}
```

No exemplo a seguir, vemos como consumir o método. Para tal, realizamos um request **GET** para a URL contendo o registro a ser manipulado **/contacts/<id>/metodo**.

rest/restful_method.php

```
<?php
require_once 'request.php';

try
{
    $location = 'http://localhost/sistema/contacts/1/name';
    print_r( request($location, 'GET') );
}
catch (Exception $e)
{
    echo 'Error: ' . $e->getMessage();
}
```

Agora, em Javascript:

```
$.ajax({
    type: 'GET',
    url: 'http://localhost/sistema/contacts/1/name',
    success: function (response) {
        console.log(response.data);
    }
});
```

Agora, em Shell

```
curl -i -X GET http://localhost/sistema/contacts/1/name
```

Retorno

Mary

8.4 Serviços RESTful seguro com token JWT

Nesta seção, vamos como criar um serviço RESTful seguro com senha global e também com tokens JWT.

Aplicação de exemplo

Uma aplicação com exemplos de serviços RESTful é disponibilizada para download no site da Adianti. Ela se chama Contacts API, e inclui serviços REST e RESTful.

Pré-requisitos

Todos os arquivos desta seção são disponíveis juntamente com o Template. Os seguintes arquivos são necessários para implementação de segurança:

- htaccess-dist
- app/config/application.ini (chaves rest_key e seed)
- app/service/auth/ApplicationAuthenticationRestService.php
- app/service/auth/ApplicationAuthenticationService.php
- rest-secure.php.dist

Obs: O arquivo `htaccess-dist` deve ser renomeado para `.htaccess`. Para o `.htaccess` ser lido corretamente, é necessário que o `AllowOverride` esteja ligado no Apache.

Introdução

A segurança na requisição RESTful no Framework pode ser tratada de duas maneiras: por meio de uma chave global (REST KEY) que autoriza qualquer requisição que a informe, ou por meio de um token JWT, que identifica o usuário tal como uma sessão.

Assim, se realizarmos comunicação entre sistemas com operações em que o usuário (sessão) não é relevante, tal como uma integração entre registros em nível de sistemas, podemos utilizar a autorização simples global (REST KEY). Já se formos realizar uma operação entre sistemas em que o perfil do usuário é relevante, tal como em casos que o Web Service utiliza a informação do usuário logado para registrar em logs de acesso, ou mesmo restringir o acesso, a autorização baseada em token JWT deve ser utilizada.

Habilitando o servidor

O primeiro passo é habilitar um servidor RESTful no Adianti Framework. Para tal, renomear o arquivo `rest-secure.php.dist` para `rest.php` que consta no diretório principal do Template e servidor estará no ar.

Autorização global

A primeira forma de segurança que demonstraremos é por meio de autorização global. Esta forma de segurança utilizará autorização por meio de uma chave. É uma forma simples de proteger o acesso a um recurso HTTP. A autorização é realizada enviando uma chave pelo header Authorization, do tipo Basic. Como o envio não é criptografado, garanta que esteja usando SSL no seu site/sistema. O primeiro passo é definir a chave global do REST no arquivo `application.ini`.

app/config/application.ini

```
[general]
rest_key = 123
```

O exemplo a seguir demonstra como realizar a chamada de um serviço RESTful por meio do método `request()`, identificando no quarto parâmetro, a autorização, que será do tipo Basic, e em seguida conterá a chave informada, que neste caso será "123".

rest/restful_secure_basic.php

```
<?php
require_once 'request.php';
try {
    // load objects by filter
    $body['limit'] = '3';
    $body['order'] = 'name';
    $body['direction'] = 'desc';

    $location = 'https://localhost/contacts/contacts';
    print_r( request($location, 'GET', $body, 'Basic 123') );
}
catch (Exception $e)
{
    echo 'Error: ' . $e->getMessage();
}
```

Neste outro exemplo, temos a mesma chamada utilizando Shell script, veja a chave de autorização sendo passada nos headers (parâmetro -H):

```
curl -X GET -H "Authorization: Basic 123" "https://localhost/contacts/contacts/1"
```

Obs: Como o envio não é criptografado, garanta que esteja usando HTTPS no seu sistema.

Autenticação por token

A segunda forma de segurança que demonstraremos é por meio de token que identifica uma sessão de usuário. O primeiro passo é configurar uma seed (semente) que será usada como chave para geração dos tokens.

app/config/application.ini

```
[general]
seed = sdf76oasdif7a
```

No tópico anterior, vimos como autorizar um request por meio de uma chave global. Porém, esta chave global não identifica um perfil de usuário do request, somente autoriza o request. Sempre que quisermos realizar operações entre sistemas em que o perfil do usuário é relevante (logs, restrição de acesso, etc), podemos usar a autorização baseada em um token JWT.

O primeiro passo para usar a autorização baseada em token é criar um token. Quem cria o token é a aplicação a partir da informação do usuário e senha. Este processo funciona como uma autenticação de login que cria uma sessão para o usuário. Neste caso, no lugar de criar uma sessão, criaremos um token criptografado que será passado como parâmetro em todos os requests que desejamos realizar com aquele usuário. O token contém a informação do usuário e também uma hora de expiração.

Para criar um token, precisamos fazer uma requisição usando a chave básica de requisição para:

```
curl -H 'Authorization: Basic 123' -i -X GET
https://localhost/contacts/auth/<user>/<senha>
```

Exemplo:

```
curl -H 'Authorization: Basic 123' -i -X GET https://localhost/contacts/auth/admin/admin
```

Para que a aplicação reconheça a rota `/auth/user/password` é necessário adicionar a linha a seguir no `.htaccess`. Esta linha irá direcionar a requisição para a classe `ApplicationAuthenticationRestService`.

.htaccess

```
RewriteRule ^auth/([A-Za-z0-9]*)([A-Za-z0-9]*)$ rest.php?
class=ApplicationAuthenticationRestService&method=getToken&login=$1&password=$2&%
{QUERY_STRING} [NC]
```

A rota `SISTEMA/auth/<user>/<senha>` recebe o usuário e senha, valida e autentica este usuário. O retorno desta requisição será um token JWT. A estrutura de um token JWT pode ser analisado pelo site jwt.io. Nele, você poderá ver o conteúdo do token, como usuário e hora de expiração. Apesar de você poder visualizar informações como usuário e hora de expiração, você não consegue alterar o token e construir um novo com um usuário diferente, em virtude do algoritmo de criptografia utilizado. Ao alterar ele, parte da assinatura que o valida se tornaria inválida.

A partir do token obtido, que identifica um usuário válido já autenticado anteriormente pelo sistema, você poderá utilizá-lo em requisições posteriores, como no exemplo a seguir. O token é informado no parâmetro do HEADER `Authorization`, após a palavra `Bearer`. O último argumento contém a rota RESTful que será acessada.

```
curl -H 'Authorization: Bearer eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c2VyIjoiYWRtaW4iLCJ1c2VyaWQiOixIiwidXNlcm5hbWUiOiJBZG1pbmlzdHJhdG9yIiwidXNlcm1haWwiOiJhZG1pbkBhZG1pbis5uZXQiLCJleHBpcmVzIjoxNTM2NzcxOTg1fQ.YPbypdCowCcjG1g08AZ-LyurrrhYUBi9HnCyufXBRpEQ' -i -X GET https://localhost/contacts/contacts/1
```

Esta requisição, além de acessar este recurso RESTful irá criar uma sessão de usuário durante esta requisição no servidor. Assim, qualquer operação neste tempo que depende do perfil do usuário irá funcionar. Ex: `TSession::getValue('login')`.

O token tem uma validade, que é definida pela classe `ApplicationAuthenticationRestService` como sendo de 3 horas o default. Depois disso, o token se torna inútil e um erro será lançado: Token expired. This operation is not allowed

O exemplo a seguir demonstra como realizar toda a operação. O primeiro request demonstra como obter o token com o usuário user, senha user. Neste caso, utilizamos a autenticação básica com chave global, que é identificada pelo parâmetro de autorização (Basic 123). O token gerado é armazenado na variável `$token`. Em seguida, é realizada uma requisição para obter o registro de contato de ID 1. Nesta segunda requisição, identificamos o token de autenticação no último parâmetro (`Bearer $token`). Esta segunda requisição criará uma sessão de usuário (user), com base no token. Assim, toda operação no lado do servidor que dependa do perfil do usuário irá funcionar.

```
<?php
require_once 'request.php';
try
{
    $location = 'http://localhost/contacts/auth/user/user';
    $token = request($location, 'GET', [], 'Basic 123');
    var_dump($token);

    $location = 'http://localhost/contacts/contacts/1';
    var_dump(request($location, 'GET', [], 'Bearer ' . $token));
}
catch (Exception $e)
{
    echo 'Error: ' . $e->getMessage();
}
```

8.5 Manipulando usuários por REST

O Template já acompanha uma classe de serviço que disponibiliza operações para usuários. Com os serviços RESTful já disponíveis, você pode criar usuários, excluir, adicionar usuários em grupo, alterar, listar, dentre outros.

Inicialmente, você precisa habilitar o servidor `rest.php`, e também as rotas REST no arquivo `.htaccess`:

```
mv htaccess-dist .htaccess
mv rest-secure.php.dist rest.php
```

Em seguida, será necessário definir uma chave global `rest_key` no `application.ini`:

app/config/application.ini

```
[general]
rest_key = 123
```

Neste primeiro exemplo, demonstramos como criar um usuário, e colocar este usuário no grupo 2 (Standard). Em primeiro lugar, fazemos uma requisição para descobrir se o usuário já existe. Caso ele não exista, fazemos uma requisição para cadastrá-lo, e em seguida outra para inseri-lo no grupo 2 (Standard).

rest/user-create.php

```
<?php
require_once 'request.php';

try {
    $body['filters'] = [ ['login', '=', 'pedro'] ];
    $location = 'http://localhost/template/users';
    $user = request($location, 'GET', $body, 'Basic 123');

    if (!$user) // se não encontrado
    {
        $body = ['name' => 'Pedro paulo',
                 'login' => 'pedro',
                 'password' => md5('123'),
                 'email' => 'pedro.paulo@teste.com',
                 'active' => '1'];
        $location = 'http://localhost/template/users';
        // insere usuário
        $data = request($location, 'POST', $body, 'Basic 123');
        print_r($data);

        $body = ['system_user_id' => $data->id,
                 'system_group_id' => '2'];
        $location = 'http://localhost/template/user-groups';

        // insere usuário no grupo
        print_r( request($location, 'POST', $body, 'Basic 123') );
    }
}
catch (Exception $e) {
    print $e->getMessage();
}
```

Neste segundo exemplo, fazemos uma requisição para alterar o nome do usuário 3, que já existe. Logo se trata de uma alteração. Veja que na URL identificamos o ID do usuário. A operação PUT também identifica que é uma alteração.

rest/user-update.php

```
<?php
require_once 'request.php';

try
{
    $body['name'] = 'Pedro Paulo - changed';
    $location = 'http://localhost/template/users/3';
    request($location, 'PUT', $body, 'Basic 123');
}
catch (Exception $e)
{
    print $e->getMessage();
}
```

Neste novo exemplo, fazemos uma requisição para obter os dados do usuário 2. Neste caso, utilizamos uma operação do tipo GET.

rest/user-get.php

```
<?php
require_once 'request.php';

try
{
    $location = 'http://localhost/template/users/2';
    $user = request($location, 'GET', [], 'Basic 123');

    print_r($user);
}
catch (Exception $e)
{
    print $e->getMessage();
}
```

Veja a seguir o retorno da operação na forma de objeto.

retorno

```
stdClass Object
(
    [id] => 2
    [name] => User
    [login] => user
    [email] => user@user.net
    [system_unit_id] =>
    [active] => Y
)
```

Neste próximo, temos a instrução `DELETE` para excluir o usuário de ID 3.

rest/user-delete.php

```
<?php
require_once 'request.php';

try
{
    $location = 'http://localhost/template/users/3';
    request($location, 'DELETE', [], 'Basic 123');
}
catch (Exception $e)
{
    print $e->getMessage();
}
```

Por fim, temos uma operação de GET para a rota `/users`, que traz todos usuários.

rest/user-list.php

```
<?php
require_once 'request.php';

try
{
    $body['order'] = 'id';
    $body['direction'] = 'asc';
    $location = 'http://localhost/template/users';
    $users = request($location, 'GET', $body, 'Basic 123');

    print_r($users);
}
catch (Exception $e)
{
    print $e->getMessage();
}
```

retorno

```
Array
(
    [0] => stdClass Object
        (
            [id] => 1
            [name] => Administrator
            [login] => admin
            [email] => admin@admin.net
            [system_unit_id] =>
            [active] => Y
        )
    [1] => stdClass Object
        (
            [id] => 2
            [name] => User
            [login] => user
            [email] => user@user.net
            [system_unit_id] =>
            [active] => Y
        )
)
```

8.6 Envio de emails

Para enviar e-mails, o Adianti Framework possui uma classe chamada `MailService`, que é uma classe de serviço que internamente se comunica com a classe `TMail`, que nada mais é do que uma fachada para a classe `PHPMailer`. O objetivo da classe `MailService` é oferecer ao desenvolvedor uma interface estável e independente de mudanças na biblioteca utilizada (neste caso, a `PHPMailer`). Dessa forma, poderíamos inclusive mudar a biblioteca de envio de e-mails por outra, sem afetar a aplicação, uma vez que a aplicação é dependente somente de `TMail`, uma classe leve e com uma implementação pequena.

Para enviar e-mails é necessário algumas configurações como: servidor, usuário, senha, autenticação smtp, dentre outros. Para configurar estes detalhes, basta usar o formulário de preferências do sistema, já apresentado quando abordamos o Template.

O trecho de código a seguir demonstra como o envio de e-mails pode ser efetuado. Basta informarmos o destinatário em forma de string contendo o e-mail ou um vetor de strings com vários e-mails, em seguida o assunto, depois a mensagem, e por último o tipo de conteúdo, que pode ser `text` ou `html`.

A classe de serviço internamente utilizará as configurações do sistema, armazenadas no formulário de preferências do sistema. Lá são configuradas variáveis como host, usuário, senha porta e outras informações para envio SMTP.

Exemplo de envio de e-mail

```
<?php
try
{
    MailService::send( $destinatario, $assunto, $mensagem, 'text' );
}
catch (Exception $e)
{
    new TMessage('error', $e->getMessage());
}
```

8.7 Pacotes Composer

Ao construir uma aplicação é muito comum precisarmos integrar bibliotecas de terceiros. O Framework não precisa e não deve oferecer de maneira embarcada todo e qualquer tipo de biblioteca, mas deve oferecer um meio de integrá-las facilmente à aplicação. O Adianti Framework já vem com algumas bibliotecas básicas e comuns para aplicações de negócios como a PHPMailer (envio de e-mails), a DomPdf (processamento de PDF), a pQuery (manipulação de HTML), BarcodeGenerator (Geração de códigos de barras) BaconQrCode (geração de QRCodes), e a PHPRtfLite (Geração de documentos RTF). Caso essas bibliotecas fossem instaladas separadamente, os seguintes comandos seriam necessários, mas elas já fazem parte da distribuição.

Bibliotecas pré-instaladas

```
composer require phpmailer/phpmailer
composer require dompdf/dompdf
composer require tburry/pquery
composer require picqer/php-barcode-generator
composer require bacon/bacon-qr-code
composer require phprtflite/phprtflite
```

Para demonstrar a integração com novas bibliotecas, vamos rodar esse comando a seguir, para instalar uma biblioteca nova, a Faker, cujo objetivo é gerar nomes de pessoas e endereços aleatórios

Outras bibliotecas

```
composer require fzaninotto/faker
```

Todas bibliotecas de terceiros são instaladas na pasta `/vendor`, e você pode acompanhar no arquivo `composer.json` as bibliotecas instaladas, bem como suas respectivas versões. Este arquivo é reescrito a cada biblioteca nova instalada.

composer.json

```
{
    "require": {
        "fzaninotto/faker": "^1.7",
        "phpmailer/phpmailer": "^6.0",
        "tburry/pquery": "^1.1",
        "picqer/php-barcode-generator": "^0.2.2",
        "dompdf/dompdf": "^0.8.1",
        "bacon/bacon-qr-code": "^1.0",
        "phprtflite/phprtflite": "^1.3"
    }
}
```

A partir do momento em que temos a biblioteca instalada, podemos fazer uso imediato em uma classe de controle. O exemplo a seguir demonstra a utilização da biblioteca Faker, cujo objetivo é a geração de nomes e endereços aleatórios para alimentar bases de dados de teste, por exemplo.

app/control/extras/FakerView.php

```
<?php
class FakerView extends TPage
{
    public function __construct()
    {
        parent::__construct();

        // criação do objeto Faker
        $faker = Faker\Factory::create();

        $output = '';
        $output .= '<b>Title</b>: ' . $faker->title . '<br>';
        $output .= '<b>Name</b>: ' . $faker->name . '<br>';
        $output .= '<b>State</b>: ' . $faker->state . '<br>';
        $output .= '<b>State Abbr</b>: ' . $faker->stateAbbr . '<br>';
        $output .= '<b>City</b>: ' . $faker->city . '<br>';
        $output .= '<b>Street Name</b>: ' . $faker->streetName . '<br>';
        $output .= '<b>Number</b>: ' . $faker->buildingNumber . '<br>';
        $output .= '<b>Country</b>: ' . $faker->country . '<br>';

        $panel = new TPanelGroup('Faker');
        $panel->add($output);
        parent::add($panel);
    }
}
```

Na figura a seguir, podemos conferir o resultado do programa.

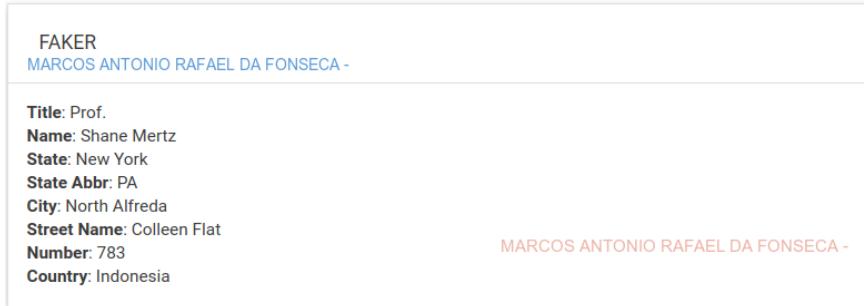


Figura 180 Aplicação rodando com a biblioteca Faker

Bibliotecas úteis

Outras bibliotecas úteis para o desenvolvimento de aplicações de negócios:

Integração com PagSeguro

```
composer require pagseguro/pagseguro-php-sdk
```

Geração de NFE

```
composer require nfephp-org/sped-nfe
```

Remessa e retorno CNAB

```
composer require andersondanilo/cnab_php
```

8.8 PWA Manifest

O Template do Adianti Framework já acompanha o arquivo de manifesto PWA no diretório raiz (`manifest.json`). Este arquivo já incluído pelos arquivos do tema (`login.html`, `public.html` e `layout.html`). Este arquivo define algumas características da aplicação quando esta rodar em dispositivos móveis, como cor do tema, ícone, etc.

Dessa forma, quando você navegar para a aplicação usando no mobile um navegador como o Chrome, bastará utilizar a opção “Adicionar a tela inicial” por meio do menu de contexto, para que seja criado um atalho na forma de ícone na tela inicial (home) do smartphone. A aplicação abrirá encapsulada na forma de aplicativo.

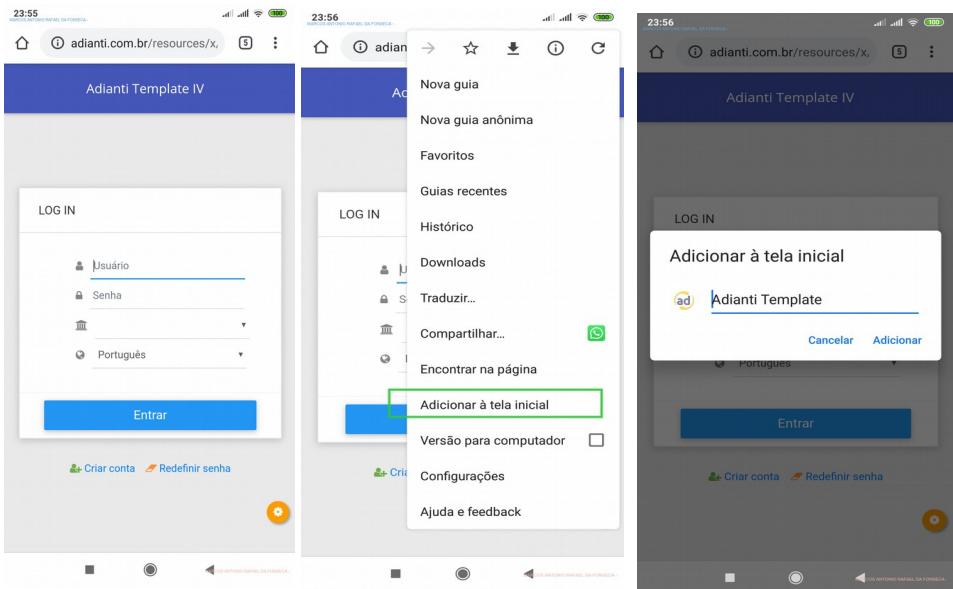


Figura 181 Adicionando a aplicação à tela inicial no mobile

Conclusão

Espero que essa não seja a conclusão de um trabalho, mas o início de um grande projeto. Da minha parte, posso naturalmente lhe adiantar que o Adianti Framework terá novas versões, e este livro, novas edições. Para manter-se atualizado, não deixe de acessar nossa comunidade e assinar a nossa lista de discussões em www.adianti.com.br.

Pablo Dall'Oglio

<pablo@dalloglio.net>

Suporte

No site da Adianti (www.adianti.com.br), você encontra nossa comunidade. Lá, poderá interagir por meio de nosso fórum de discussões, e também poderá inscrever-se em nossa lista de discussões. Além disso, a Adianti oferece serviços de treinamento e consultoria.

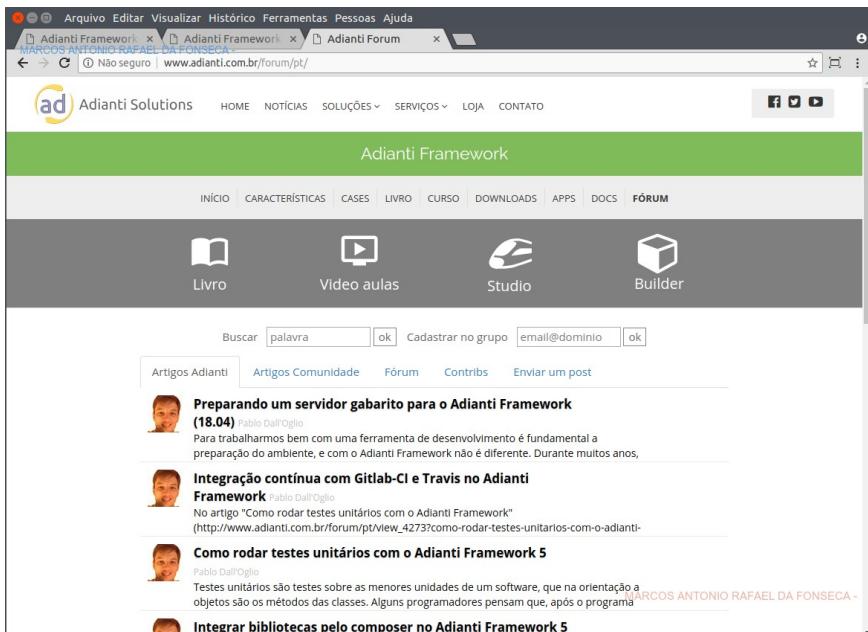


Figura 182 Comunidade no site da Adianti

Treinamento

Devido à demanda por treinamentos a partir de vários pontos do país, a dificuldade em atender a todos por causa da distância, foi lançado o treinamento completo sobre o Adianti Framework no formato de videoaulas, que pode ser adquirido diretamente no site da Adianti, em sua loja virtual (<http://www.adianti.com.br/store>).

A partir da compra das videoaulas, você recebe um link onde terá acesso exclusivo ao conteúdo. As aulas foram todas gravadas no formato HD pelo próprio criador do framework. São mais de 20 horas de curso, onde você aprenderá a criar uma aplicação, passando pela parte de banco de dados, modelos, persistência, apresentação, integração com banco de dados, criação de controle de permissões de acesso, alteração de layout, dentre outros. O treinamento em videoaulas, bem como o livro sobre o framework, formam o material mais completo disponível sobre o Adianti Framework.

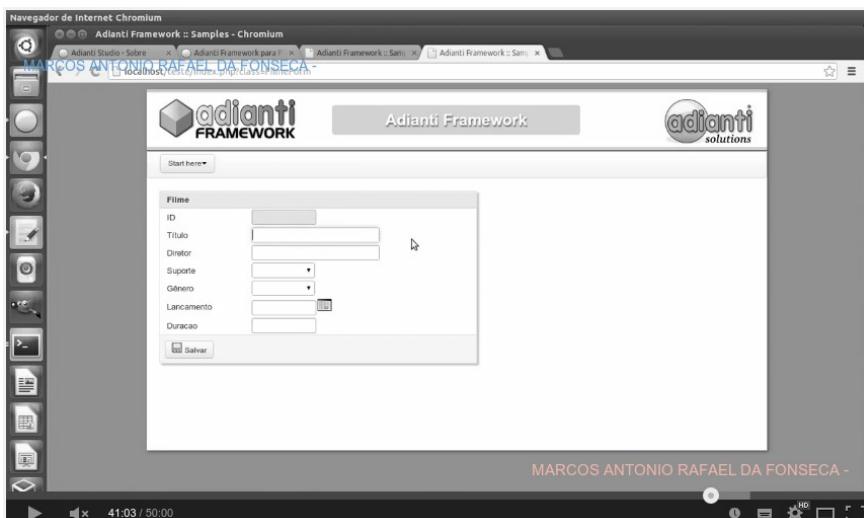


Figura 183 Videoaulas sobre o Adianti Framework

Adjanti Studio

O Adianti Studio é uma IDE Desktop para desenvolvimento PHP que possui uma versão Professional. O Adianti Studio Pro automatiza a criação de código-fonte para o Adianti Framework, tornando muito mais ágil o desenvolvimento de novos sistemas. O Adianti Studio Pro oferece diversos assistentes para geração de código, que são telas passo a passo que atuam sobre uma tabela e geram código, como:

- Assistente para criação de formulários de edição e consulta;
 - Assistente para criação de listagens com filtros;
 - Assistente para criação de formulários mestre/detalhe;
 - Assistente para criação de relatórios tabulares.

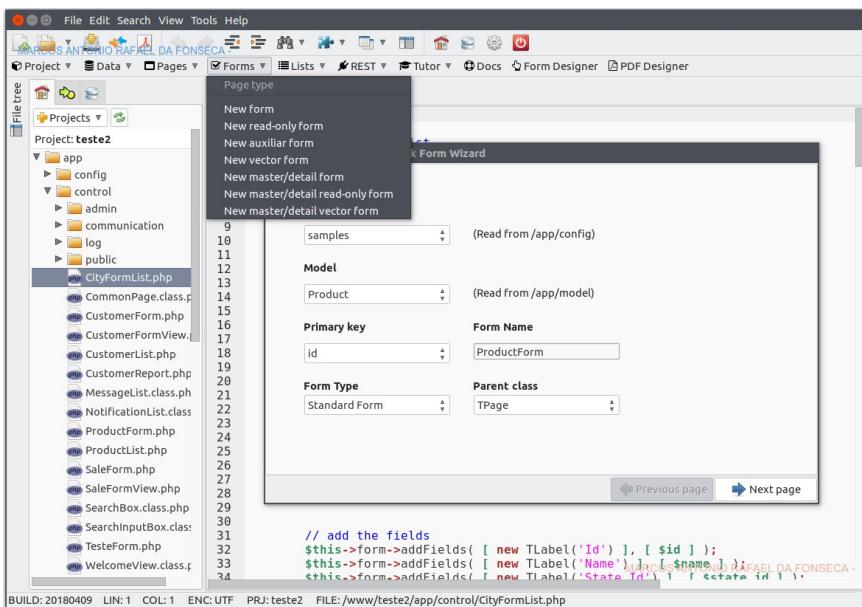


Figura 184 Adianti Studio

Adianti Builder

O Adianti Builder é um construtor online de aplicações, que vai desde a modelagem do banco, até o projeto de telas de formulários, datagrids, relatórios, documentos, gráficos, e outros. Permite realizar a maioria das definições de maneira visual, e também com pontos de edição de código-fonte.

Após modelagem dos dados, é possível criar facilmente formulários, listagens, documentos, relatórios, gráficos, calendários, utilizando definições visuais, e funções de clique e arraste. É possível definir o comportamento de ações como botões de formulário, ações de datagrids, criar códigos personalizados, e muito mais.

O Adianti Builder, além de um construtor visual, também é um editor de código online. Ao final, o usuário poderá realizar download da aplicação, ou enviar diretamente para um de seus servidores por meio da função de deploy.

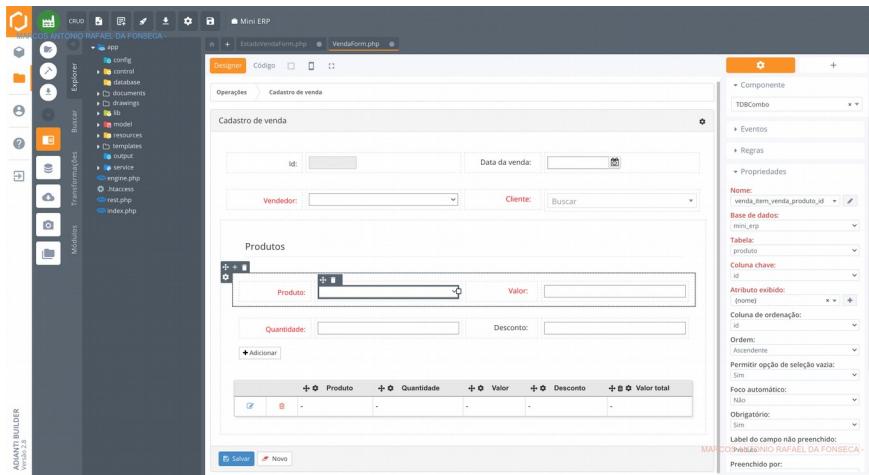


Figura 185 Adianti Builder

Canais Adianti

Não deixe de acompanhar as novidades por nossos canais:

<https://www.facebook.com/adiantisolutions>

<https://twitter.com/adiantisolution>

<https://www.youtube.com/AdiantiSolutions>

Do mesmo autor

