



Este livro objetiva introduzir o leitor nos principais conceitos e funcionalidades básicas do Titan Framework, apresentando, de forma lúdica, os ensinamentos fundamentais para o desenvolvimento de aplicações Web por meio desta ferramenta. Caso seja um desenvolvedor experiente, recomenda-se a leitura do volume “**Cookbook Master Chef**”, que apresenta funcionalidades avançadas e detalha a API de desenvolvimento de componentes específicos.

Sumário

I. O Titan Framework

1.1 Introdução

1.2 Arquitetura

1.3 Estrutura da Aplicação

1.4 Tipos de Usuários, Grupos e Permissões

1.5 Repositório Global

1.5.1 Componentes Globais

global.archive, global.contact, global.generic, global.group, global.home, global.poller, global.simple, global.userPrivate, global.userPublic e global.userProtected

1.5.2 Tipos Globais

Phrase, Cpf, Cnpj, Cep, Email, Phone, Login, Url, Enum, Radio, CheckBox, Twitter, Color, Slug, TimeZone, Integer, Amount, Boolean, File, Double, Money, Select, Cascade, City, State, Multiply, Collection, Document, Date e Time

1.6 Boas Práticas de Programação

1.6.1 Convenções para o Banco de Dados

1.6.2 Convenções de Código

II. Criando a Primeira Aplicação

2.1 Ambiente de Desenvolvimento

2.2 Instanciação da Aplicação

2.3 Criação de Tipos e Componentes

2.3.1 Criando um Componente Derivado

2.3.2 Criando Tipos Derivados

2.4 Publicando a Aplicação

2.4.1 Configurando um Ambiente de Produção

2.4.2 Auto-Deploy e Database Migrations

I. O Titan Framework

1.1 Introdução

O Titan é um *framework* para instanciamento de Sistemas de Gerenciamento de Conteúdo (do inglês, *Content Management Systems* - CMS), aplicações Web utilizadas para criar, editar, gerenciar e publicar conteúdo de forma consistentemente organizada, permitindo que o mesmo seja modificado, removido e adicionado com facilidade. Foi implementado em **PHP**¹ e utiliza o **PostgreSQL**² como banco de dados principal.

Uma das principais características do Titan é possuir um conjunto de código imutável e legado denominado “**núcleo**” (do inglês, **core**) que é automaticamente atualizado, mesmo em produção, garantindo que todas as instâncias permaneçam seguras e confiáveis.

Outro diferencial importante é sua arquitetura única, com foco total em reúso. O Titan possui um repositório de artefatos parametrizáveis (com componentes, tipos de dados, *templates* de código, elementos de *layout*, ferramentas e *drives*). Desta forma, boa parte da programação, tal como a definição de modelos de dados, pode ser feita por meio de linguagem de marcação. Esta característica auxilia muito a manutenção corretiva e evolutiva de funcionalidades da aplicação.

Como será visto, o Titan pode ser facilmente instanciado gerando um CMS pronto para uso com diversas funcionalidades. Para que o desenvolvedor implemente seus requisitos nesta aplicação Web inicial, ele faz uso dos diversos componentes do Titan ou implementa novos, caso os existentes não atendam. Na Figura 1.1 é mostrada a tela de *login* padrão do *framework*. Neste exemplo, extraído do sistema Pandora³ da Embrapa, foram ativadas diversas funcionalidades inerentes ao *framework*, tal como o acesso e registro de usuários utilizando redes sociais, a validação de documentos gerados pela instância e a autenticação por LDAP.

¹ <http://php.net/>

² <http://www.postgresql.org/>

³ <http://cloud.cnpqc.embrapa.br/pandora>

ACESSE USANDO SUA REDE SOCIAL FAVORITA:



QUERO ME CADASTRAR...

JÁ SOU CADASTRADO...

Cadastrar-se como "Comunidade Externa"

AUTENTICAR DOCUMENTOS E CERTIFICADOS...

Utilizando o QR Code do documento

Utilizando informações do documento

Bem vindo ao gestor do PANDORA • Sistema Integrado de Gestão e Pesquisa!

Você precisa efetuar login para continuar.

login

Login:

Senha:

[\[Esqueci minha senha\]](#)

INSTALE NOSSO APLICATIVO MÓVEL:



Figura 1.1: Tela de *login* nativa do *framework* (disponível por padrão para todas as instâncias).

Na Figura 1.2, extraída do mesmo sistema, pode-se observar uma tela do framework exibida ao usuário logado. Trata-se, portanto, de uma seção e ação ativa naquele momento. Nesta imagem é possível observar diversos outros elementos do framework. Por exemplo, na barra superior, do lado esquerdo, está o tipo do usuário logado e os grupos de permissões aos quais ele pertence; na mesma barra, no canto superior direito, está uma coleção de ícones que dão acesso à funcionalidades globais do sistema (página inicial, perfil do usuário, sistema de notificações, idiomas, reporte de problema técnico, *backup*, manual de uso da instância e *logout*); e, no canto inferior esquerdo encontra-se o nome do usuário logado e o botão de menu, que permite que este usuário navegue pelas demais seções do sistema.

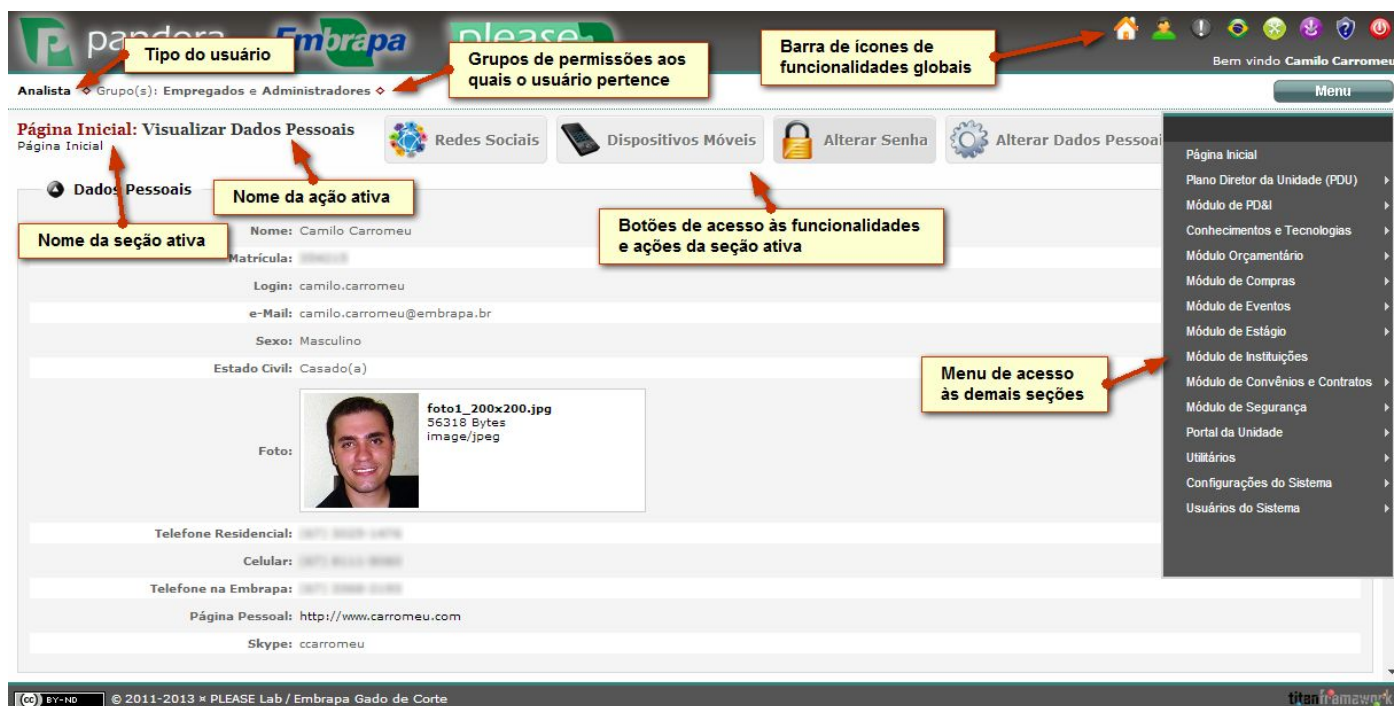


Figura 1.2: Típica tela de uma instância (após *login*).

1.2 Arquitetura

Todas as funcionalidades do Titan são organizadas em um modelo de navegação baseado em “**seções**” e “**ações**”. Uma seção é uma funcionalidade da aplicação, tal como a gestão de notícias de um site. As ações são as atividades atômicas que os usuários podem realizar naquela funcionalidade, tal como criar, editar e visualizar notícias.

As seções e ações do Titan são renderizadas em tempo de execução por meio da parametrização, utilizando XML, de código PHP. Este código de script (*server side*) fica organizado em “**componentes**” e “**motores**” (do inglês, **engines**). Assim, cada seção da instância precisa estar relacionada a um componente (que irá renderizá-la) e cada ação desta seção precisa estar relacionada a uma *engine* deste componente. Na Figura 1.3 é mostrada esta relação.

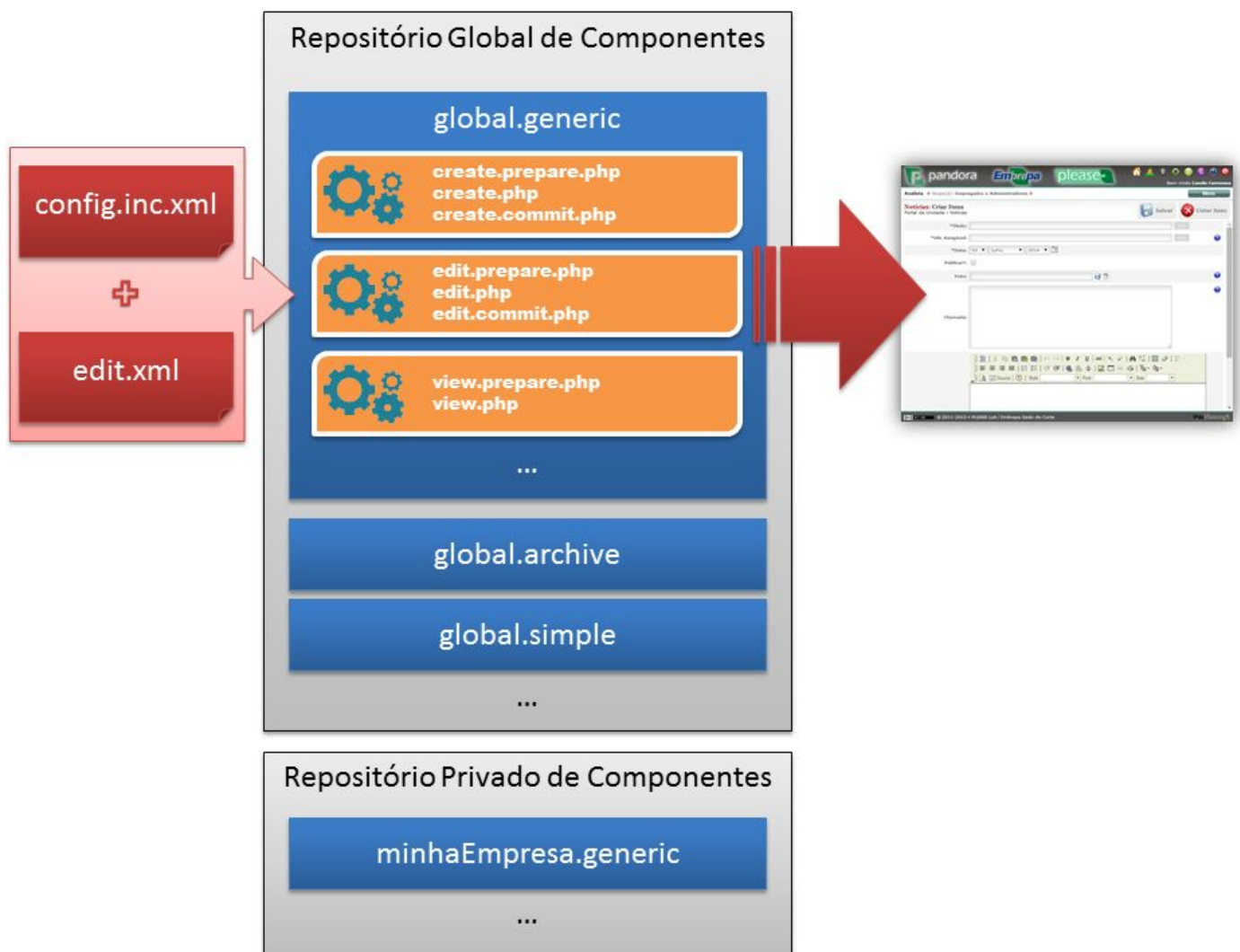


Figura 1.3: Esquema geral da renderização de seções e ações por meio de componentes e *engines*.

Neste caso, o componente denominado “**global.generic**” recebe um arquivo “**config.inc.xml**”, que possui a definição de todas as ações da seção que está sendo gerada. Para a ação ativa, está sendo utilizado a *engine* denominada “**edit**”, que está sendo parametrizada pelo arquivo “**edit.xml**”. Repare que cada *engine* é formada por até três *scripts* PHP:

- **[engine].prepare.php:** Responsável pela camada de Modelo (do inglês, *Model*), irá carregar os dados das entidades consultando a camada de persistência e fornecê-los à camada de visualização;
- **[engine].php:** Responsável pela camada de Visualização (do inglês, *View*), irá exibir ao usuário os dados das entidades; e
- **[engine].commit.php:** Responsável pela camada de Controle (do inglês, *Controller*), irá

processar as requisições validando e salvando dados submetidos na camada de persistência e direcionando o fluxo de controle para a próxima seção e ação.

De forma geral, com esta estrutura o Titan é capaz de gerar, em tempo de execução, todas as seções e ações do sistema. É importante compreender este conceito para acompanhar as próximas etapas deste documento.

1.3 Estrutura da Aplicação

Esta seção apresenta uma visão geral da estrutura comum de uma instância do Titan. Observe com atenção a Figura 1.4, nela é mostrada a estrutura de arquivos e diretórios de uma instância típica. O arquivo “**titan.php**”, na raiz, é o *bootstrap* da aplicação, ele irá carregar o núcleo do Titan a partir do caminho declarado em “**configure/titan.xml**”, o principal arquivo de configuração da instância.

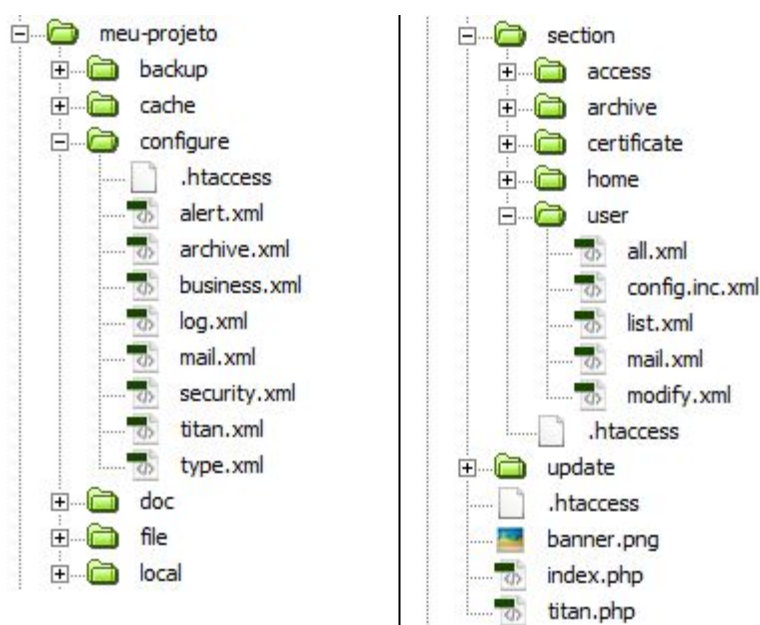


Figura 1.4: Estrutura de diretórios e arquivos de uma instância do Titan.

Uma vez que o núcleo do Titan esteja carregado, ele irá ler o arquivo “**configure/business.xml**”, onde estão declaradas todas as seções da instância (cada uma referenciando o componente que irá renderizá-la). O Titan monta o menu de navegação a partir das seções declaradas neste arquivo, levando em consideração as permissões de acesso do usuário. Para saber quais ações a seção terá, o Titan procura dentro da pasta “**section**”, que está na raiz,

uma pasta homônima à seção. Uma vez encontrada, ele irá procurar um arquivo chamado “**config.inc.xml**”, que têm a lista de todas as ações disponíveis naquela seção (cada uma referenciando a *engine*, dentro do componente da seção, que irá renderizá-la).

Por exemplo, na instância mostrada na Figura 4 há uma seção denominada “**user**”. Quando o usuário acessa esta seção pelo menu da aplicação, o Titan irá carregar as ações declaradas no arquivo “**section/user/config.inc.xml**”. No estudo de caso presente neste livro, estes arquivos serão vistos com mais profundidade.

Na estrutura mostrada, os outros diretórios são:

- **backup:** Local onde serão gravados os arquivos gerados pela funcionalidade de “**backup on demand**”, um tópico avançado apresentado no “**Cookbook Master Chef**”.
- **cache:** É o diretório (de presença obrigatória) onde o Titan irá gerar uma série de arquivos de *cache* necessários ao funcionamento e otimização de desempenho da instância.
- **configure:** Onde estão os arquivos globais de configuração. Os principais são:
 - **titan.xml:** principal arquivo de configuração onde estão dados essenciais à instância, tal como o a configuração do Banco de Dados;
 - **business.xml:** onde se declara as instâncias da aplicação e qual componente irá renderizar cada uma;
 - **mail.xml:** onde são configurados os e-mails do sistemas, tal como o que é enviado quando o usuário se registra ou quando ele utiliza a opção “Esqueci minha senha”;
 - **security.xml:** neste arquivo são declarados os 'tipos de usuários' que acessarão o sistema (veja a próxima Seção), para cada tipo deverá haver uma seção homônima que será responsável pela gestão dos usuários daquele tipo cadastrados;
 - **type.xml:** neste arquivo o usuário declara os seus tipos locais, ou seja, aqueles que ele implementou para sobrecarregar as funcionalidades dos tipos nativos; e
 - **archive.xml:** neste XML são listados todos os tipos de arquivos de *upload* aceitos pelo sistema.
- **doc:** É o diretório onde podem ser colocados arquivos para sobrescrever os criados pela funcionalidade de geração automática de manual de usuário, um tópico avançado que será apresentada no “**Cookbook Master Chef**”.

- **file:** É o diretório (de presença obrigatória) onde o Titan irá armazenar os arquivos de *upload* do sistema, ou seja, àqueles que foram enviados pelos usuários da aplicação.
- **local:** É o repositório de artefatos local da instância, onde ficará, por exemplo, os tipos e componentes específicos criados pelo desenvolvedor. Não é um diretório de presença obrigatória e deverá ser criado apenas caso existam artefatos específicos e o desenvolvedor queira seguir esta convenção de nome.
- **section:** Conforme explicado, é onde ficam os diretórios com os arquivos XML que parametrizam cada seção da aplicação.
- **update:** Diretório onde são colocados os arquivos para auto-update do Titan. Esta funcionalidade será vista no final deste livro.

TODO: Explicação dos principais arquivos XML do Titan.

1.4 Tipos de Usuários, Grupos e Permissões

O Titan permite que sejam configurados, pelo desenvolvedor, diversos “**tipos de usuários**” que ficarão disponíveis na instância. Um “**tipo de usuário**” é a delimitação de um modelo de dados para uma fatia de usuários que acessam o sistema. Assim, caso tenhamos, por exemplo, um sistema de controle acadêmico que será acessado por “professores” e “alunos” podemos esperar que, apesar das entidades que representam cada um destes atores compartilharem diversos campos (tal como “nome” ou “telefone”), haverá campos específicos no perfil de cada um destes atores (tal como “número da matrícula” e “ano” para o aluno e “salário” para o professor). Assim, a instância deverá ter sido configurada pelo desenvolvedor com estes dois tipos de usuários. Repare que a especificação dos tipos de usuários deverá ser feita no desenvolvimento da aplicação, por meio de arquivos de marcação XML. Cada usuário da instância poderá estar associado a apenas um tipo de usuário.

TODO: Exemplificar a criação de tipos no 'security.xml'.

Por padrão, cada ação de cada seção do *framework* possui uma **permissão de acesso**. Para que o usuário visualize no menu uma determinada seção e possa acessá-la, ele deverá ter permissão de uso de pelo menos uma ação naquela seção. É possível alocar as permissões em **grupos** e associar usuários a eles. Ou seja, uma permissão não é associada diretamente ao usuário

e sim indiretamente por meio dos grupos aos quais aquele usuário pertence. Na Figura 1.5 é possível visualizar a tela em que se associa permissões a um determinado **grupo**.



Figura 1.5: Associação de permissões à grupos de usuários.

Por fim, é possível associar aos **tipos de usuários**, quais **grupos** deverão ser associados no momento em que um usuário daquele tipo é cadastrado ou se registra no sistema. Em resumo, os tipos de usuários serão, portanto, responsáveis pelo modelo de dados do usuário, enquanto os grupos estabelecem quais permissões o usuário possui.

1.5 Repositório Global

O Titan possui um repositório de ativos próprio, que possui diversos conjuntos de artefatos que podem ser reutilizados para diferentes propósitos. Os artefatos contidos neste repositório global, presente no núcleo do Titan, são denominados **artefatos nativos**. No âmbito deste livro, veremos os dois conjuntos principais, ou seja, os mais importantes para qualquer instância de algum aplicativo construído com o *framework*:

- **Componentes:** Um componente, conforme já explicado na Seção “1.2 Arquitetura”, é o conjunto de *scripts* PHP que é parametrizado para instanciar uma seção. Os componentes nativos do Titan possuem um alto grau de parametrização, permitindo criar seções com comportamento similar mas com modelos de dados completamente distintos. Para ilustrar, pode-se criar com o componente “global.generic” uma seção que gerencia notícias de um portal e uma seção que gerencia informações institucionais relacionadas ao portal, mesmo que não possuam nenhum campo semelhante.
- **Tipos:** Como o intuito do Titan é instanciar sistemas CMS, normalmente uma seção contém formulários para a criação, visualização, edição e deleção de tuplas das entidades do Banco de Dados. No Titan a camada de dados da aplicação é, em parte, implementada por meio de arquivos XML. Estes arquivos parametrizam classes da API do *framework* instanciando, em tempo de execução, os objetos do modelo de dados (do inglês, *Data Access Object* - DAO). Estes objetos são passados à camada de visualização, permitindo representar os formulários supracitados. Estes XMLs são compostos por uma série de “**campos**” (do inglês, ***fields***) que, por sua vez, mapeiam um “**tipo**” (do inglês, ***type***) do *framework*. Os tipos são, portanto, um conjunto de *scripts* PHP que são parametrizados por estas entradas (*tags*) do XML, mapeando os atributos nestes objetos do modelos de dados.

A seguir, é apresentada a lista dos **componentes** e **tipos** nativos do repositório do Titan Framework.

1.5.1 Componentes Globais

TODO: Explicação detalhada de cada componente (explicar o uso/ativação de cada *engine* e os XMLs obrigatórios).

global.archive

Componente utilizado para criar uma seção de gerenciamento de *uploads* da instância, com listagem de arquivos já enviados e funcionalidades de gerenciamento destes arquivos.

global.contact

Componente utilizado para gerenciar mensagens enviadas através de algum formulário de contato com o usuário. Oferecendo que o usuário possa responder essas mensagens através da instância do Titan.

global.generic

Componente utilizado para criar seções do tipo CRUD (acrônimo de *Create, Retrieve, Update and Delete*).

global.group

Componente utilizado para gerenciar os grupos de usuários e suas permissões dentro de uma instância do Titan.

global.home

Componente utilizado para criar uma página inicial de uma instância do Titan, tal como um *dashboard* sobre a instância.

global.poller

Componente utilizado para gerenciar enquetes que pode ser usadas no portal que é gerenciado por uma instância do Titan.

global.simple

Componente utilizado para inserção de informações estáticas.

global.userPrivate

Componente utilizado para gerenciar usuários que são de tipos “privados” da instância, ou seja, usuários que somente podem ser cadastrados por outros usuários, logados na instância e com permissões para tal.

global.userPublic

Componente utilizado para gerenciar usuários que são de tipos “públicos” da instância, ou seja, usuários que podem utilizar um formulário disponível publicamente para efetuar cadastro na

instância e passar, imediatamente, a utilizá-la.

global.userProtected

Componente utilizado para gerenciar usuários que são de tipos “protegidos” da instância, ou seja, usuários que podem utilizar um formulário disponível publicamente para efetuar cadastro na instância, mas necessitam ser validados por outros usuários da instância para acessá-la.

1.5.2 Tipos Globais

Os tipos do Titan são organizados em uma relação de pai e filho (classe derivada). Assim, o tipo “**global.Amount**” é derivado do tipo “**global.Integer**”, ou seja, o primeiro utiliza o segundo como base, mas implementa comportamentos específicos. Por exemplo, no caso do “global.Amount” o campo que aparecerá no formulário, bem como o número mostrado em uma ação de visualização, serão formatados por meio de uma máscara que separa os 'milhares' por ponto.

Esta característica permite que um desenvolvedor estenda facilmente os tipos nativos. Por exemplo, ao criar um novo tipo denominado “ProjectName.MyType” que herda o tipo nativo “global.Radio”, este novo tipo terá todas as funcionalidades do tipo nativo. Se observarmos, entretanto, o tipo nativo “global.Radio”, veremos que ele herda o tipo “global.Enum”. Este último, por sua vez, herda o tipo “global.Phrase”. Assim, podemos concluir que o tipo “ProjectName.MyType” têm características de todos estes tipos, que foram sobreescritas hierarquicamente na relação hereditária. Na Figura 1.6 é mostrada esta relação hereditária entre os tipos nativos do Titan.

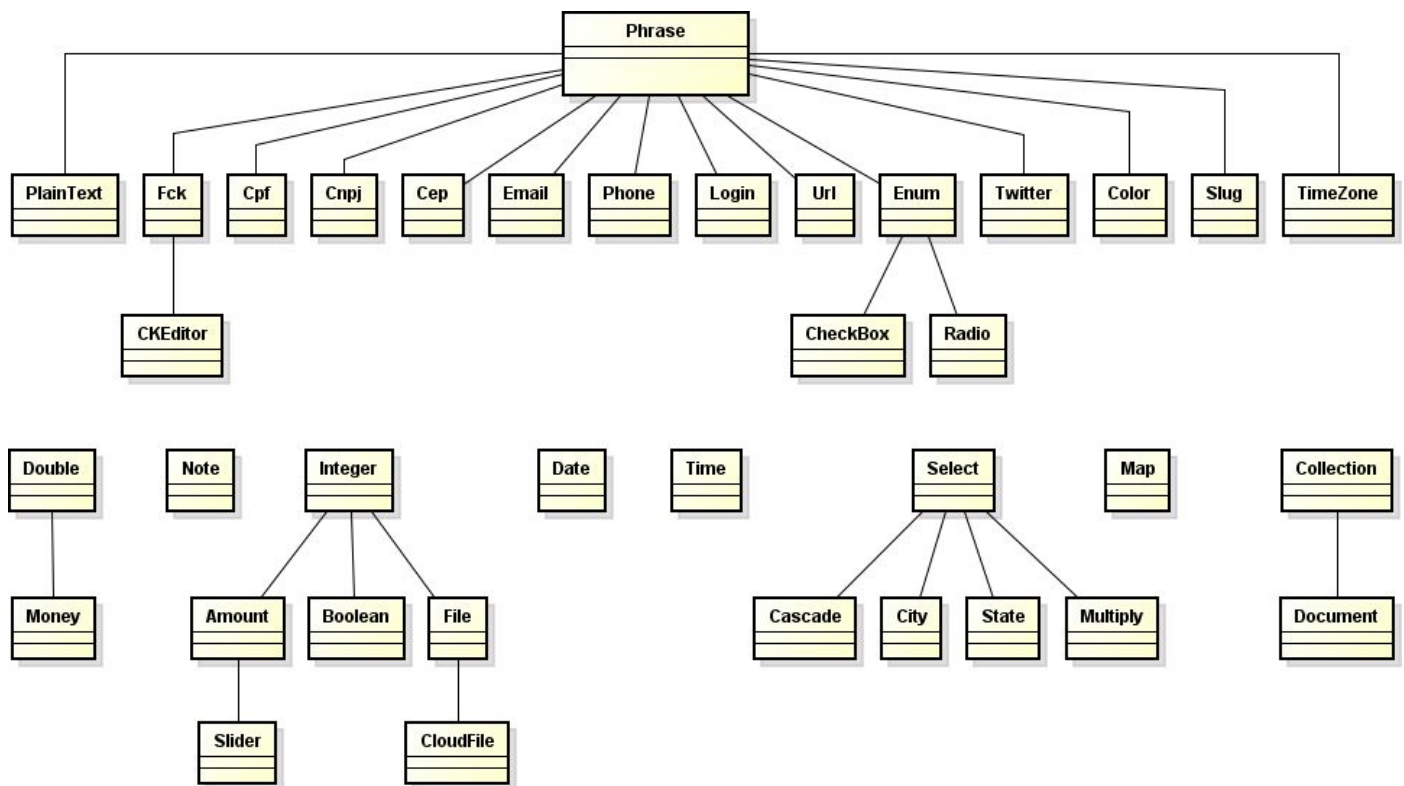


Figura 1.6: Relação hereditária entre os tipos nativos do Titan.

Repare que no Titan para fazer até mesmo pequenas distinções comportamentais como a do tipo “global.Amount” em relação ao “global.Integer”, que fogem das configurações nativas disponíveis nos atributos de parametrização no XML, é necessário estender os tipos existentes.

Todos os tipos do Titan são mapeados em arquivos de marcação XML com o uso da tag “**field**”. Todos eles compartilham alguns atributos básicos, mas podem também ter atributos específicos. Por exemplo, o tipo “Double” possui um atributo denominado “*precision*”, onde o desenvolvedor pode estipular qual o número de casas decimais do campo. Os atributos padrões, comuns a todos os tipos, são:

- **type:** nome do tipo;
- **column:** nome da coluna na tabela no banco de dados;
- **label:** rótulo do campo;
- **help:** informação adicional sobre o campo, acessível ao usuário por um ícone de ajuda ao lado direito do campo;
- **tip:** dica de preenchimento do campo, que aparece em um fundo azul imediatamente após a

área de preenchimento do campo;

- **table:** tabela no banco de dados a que se refere o campo (por padrão utiliza a declarada no formulário);
- **value:** valor padrão a ser atribuído ao campo;
- **id:** um identificador único, que permite obter o objeto instanciado deste campo no código da aplicação (por padrão o Titan preenche este atributo com um valor único);
- **required:** aceita 'true' ou 'false' para determinar se o preenchimento do campo é obrigatório ou não (por padrão é 'false');
- **unique:** aceita 'true' ou 'false' para determinar se o valor deve ser único ou não (por padrão é 'false');
- **read-only:** aceita 'true' ou 'false' configurando para que o campo não seja editável em determinado formulário (por padrão é 'false');
- **restrict:** utilizado em conjunto com o sistema de controle de acesso do framework, determinando se o campo poderá ser editado apenas por usuários com permissões específicas;
- **style:** permite alterar a aparência do campo utilizando CSS; e
- **doc:** informações adicionais sobre o campo para serem inseridas no manual automaticamente gerado pelo *framework*.

Desta forma, poderíamos, por exemplo, instanciar um formulário com um único campo do tipo “Phrase” utilizando os atributos padrões listados acima:

```
<form table="nome_da_tabela" primary="chave_primaria">
  <field
    type="Phrase"
    column="nome_da_coluna"
    id="_ID_UNICO_PARA_ESTES_CAMPO_"
    label="My Test Field | pt_BR: Meu Campo Teste | es_ES: Meu Campo Teste"
    value="O valor padrão deste campo é esta frase."
    required="true"
    unique="false"
    read-only="false"
    style="border-color: #900; font-weight: bold; color: #090;"
    tip="Ex.: Uma frase qualquer."
    help="Preencha uma frase a seu gosto."
    doc="Exemplo do uso dos tipos do Titan na criação de campos de formulários."
  />
</form>
```


Que resultará na renderização mostrada na Figura 1.7.



Figura 1.7: Renderização de um *field* em um formulário.

A seguir são detalhados cada um dos tipos nativos do Titan.

Phrase

O tipo “Phrase” é a base de diversos outros tipos do *framework*. Seu objetivo é possibilitar a entrada de frases (*strings*). Pode-se setar o tamanho máximo da oração por meio de um atributo específico denominado '**max-length**'. Por exemplo:

```
<field
  type="Phrase"
  column="nome_da_coluna"
  label="My Test Field | pt_BR: Meu Campo Teste | es_ES: Meu Campo Teste"
  max-length="256"
/>
```

É representado no banco de dados pelo tipo '*character varying*' (usualmente com tamanhos em potência de dois). Por exemplo:

```
ALTER TABLE tabela ADD COLUMN coluna VARCHAR(256);
```

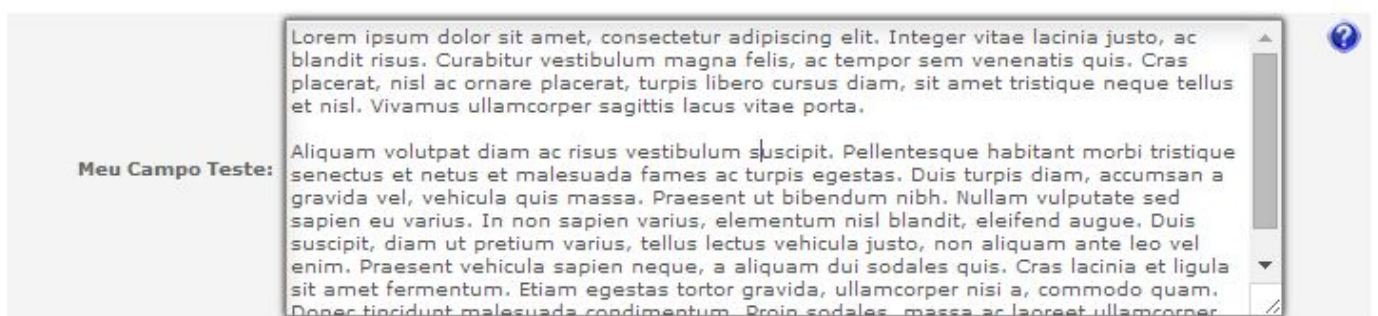


Figura 1.8: Campo de entrada de dados do tipo “PlainText”.

PlainText

O tipo “PlainText”, derivado do “Phrase”, objetiva possibilitar a entrada e representação de texto puro (*plain text*). Uma representação da entrada deste tipo de dado pode ser visualizada na Figura 1.8. É representado no banco de dados pelo tipo 'text':

```
ALTER TABLE tabela ADD COLUMN coluna TEXT;
```

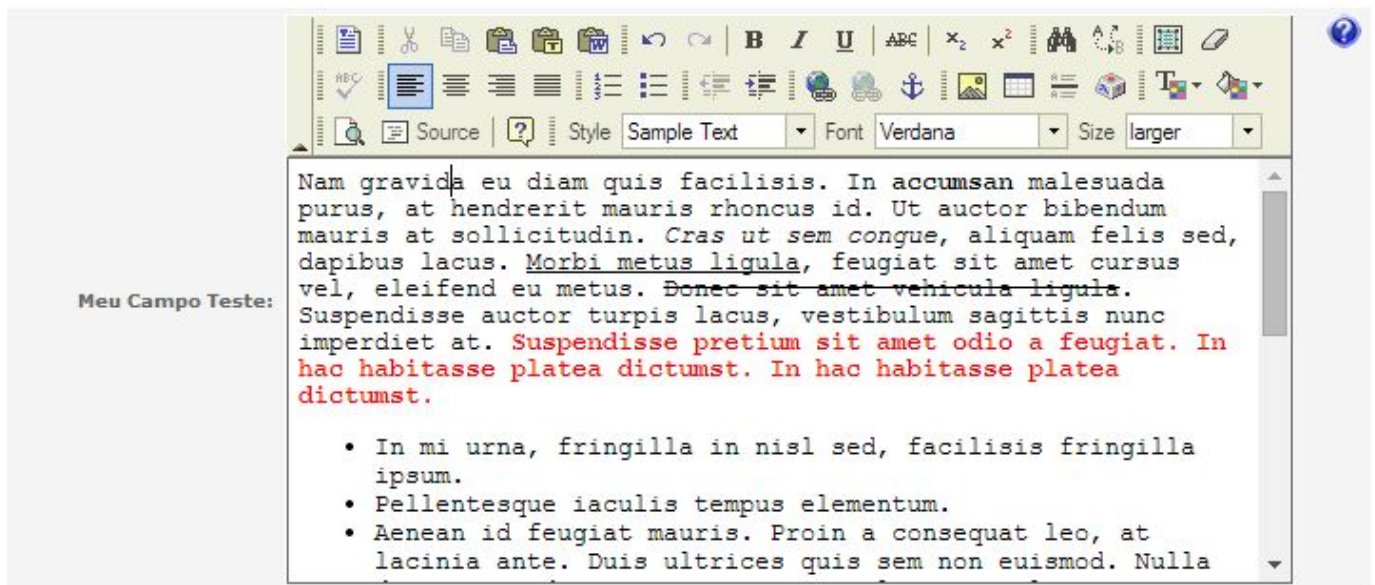


Figura 1.9: Campo de entrada de dados do tipo “Fck”.

Fck

Utilizado para possibilitar a entrada e representação de texto com formatação HTML. Oferece ao usuário um editor de textos que possibilita formatar o conteúdo de forma amigável (um editor WYSIWYG⁴). Uma representação da entrada deste tipo de dado pode ser visualizada na Figura 1.9. É representado no banco de dados pelo tipo 'text':

```
ALTER TABLE tabela ADD COLUMN coluna TEXT;
```

Cpf

Utilizado para possibilitar a entrada e representação de valores de CPF (número do

⁴ <http://pt.wikipedia.org/wiki/WYSIWYG>

Cadastro de Pessoa Física da Receita Federal). É automaticamente aplicada uma máscara para que o número fique no formato 'xxx.xxx.xxx-xx' (onde 'x' é um dígito). Somente permitirá a entrada de números de CPF válidos. É representado no banco de dados pelo tipo '*char*' de tamanho fixo com 11 (onze) caracteres:

```
ALTER TABLE tabela ADD COLUMN coluna CHAR(11);
```

Cnpj

Utilizado para possibilitar a entrada e representação de valores de CNPJ (número do Cadastro Nacional de Pessoa Jurídica da Receita Federal). É automaticamente aplicada uma máscara para que o número fique no formato 'xxx.xxx.xxx/xxxx-xx' (onde 'x' é um dígito). Somente permitirá a entrada de números de CNPJ válidos. É representado no banco de dados pelo tipo '*char*' de tamanho fixo com 15 (quinze) caracteres:

```
ALTER TABLE tabela ADD COLUMN coluna CHAR(15);
```

Cep

Utilizado para possibilitar a entrada e representação de valores de CEP (número do Código de Endereçamento Postal). É automaticamente aplicada uma máscara para que o número fique no formato 'xx.xxx-xxx' (onde 'x' é um dígito). É representado no banco de dados pelo tipo '*char*' de tamanho fixo com 8 (oito) caracteres:

```
ALTER TABLE tabela ADD COLUMN coluna CHAR(8);
```

Email

Utilizado para possibilitar a entrada e representação de e-mails válidos. No momento em que o usuário digita o e-mail, o valor de entrada é validado (inclusive com a consulta ao provedor do e-mail). Não é, portanto, recomendado o uso deste tipo em instância que não terão acesso permanente à internet. É representado no banco de dados pelo tipo '*character varying*' com 256

caracteres (tamanho máximo de um endereço de e-mail⁵):

```
ALTER TABLE tabela ADD COLUMN coluna VARCHAR(256);
```

Phone

Utilizado para possibilitar a entrada e representação de números de telefone. É automaticamente aplicada uma máscara para que o número fique no formato '(xx) xxxx-xxxx' (onde 'x' é um dígito). É representado no banco de dados pelo tipo '*char*' de tamanho fixo com 10 (dez) caracteres:

```
ALTER TABLE tabela ADD COLUMN coluna CHAR(10);
```

Login

Utilizado para possibilitar a entrada e representação de logins quando a instância é integrada à serviços de diretórios (tal como o Active Directory ou o LDAP). Permite apenas o uso de dígitos (0-9), letras minúsculas (a-z), *underscore* (_) e ponto (.). É representado no banco de dados pelo tipo '*character varying*':

```
ALTER TABLE tabela ADD COLUMN coluna VARCHAR(64) NOT NULL UNIQUE;
```

Este tipo é destinado ao uso, principalmente, em formulários de edição de dados de usuários, ou seja, tuplas da tabela mandatória '**__user**':

```
<form table="titan._user" primary="_id">
  ...
  <field
    type="Login"
    column="_login"
    label="Login"
    max-length="64"
    required="true"
    unique="true"
    help="Login utilizado para acessar o sistema."
  />
  ...
```

⁵ <http://tools.ietf.org/html/rfc5321#section-4.5.3>

</form>

Url

Utilizado para possibilitar a entrada e representação de URLs. Possui um atributo específico, denominado '**prefix**' que pode ser setado com um prefixo obrigatório para as URLs inseridas.

Por exemplo, na Figura 1.10 é mostrado o uso deste campo para que o usuário possa entrar com a URL para seu currículo na Plataforma Lattes⁶. Neste caso o atributo 'prefix' foi atribuído com "http://lattes.cnpq.br/". É importante notar que, neste caso, o atributo '**max-length**', herdado de "Phrase", foi preenchido com o valor '38', ou seja, o tamanho do conteúdo de 'prefix' somado ao tamanho máximo da entrada do usuário.




Figura 1.10: Campo de entrada de dados do tipo "Url".

O mapeamento deste tipo no formulário é mostrado abaixo:

```
<field
  type="Url"
  column="lattes"
  label="URL"
  max-length="38"
  prefix="http://lattes.cnpq.br/"
  unique="true"
/>
```

É representado no banco de dados pelo tipo '*character varying*'. Para o exemplo acima ficaria:

```
ALTER TABLE titan._user ADD COLUMN lattes VARCHAR(38) UNIQUE;
```

Enum

⁶ <http://lattes.cnpq.br>

Utilizado para possibilitar a representação de um campo com opções pré-definidas (menu *drop-down*), das quais o usuário escolhe apenas uma. Repare que os valores aceitos são definidos pelo usuário no momento em que mapeia o campo no XML (não há relacionamento com outra entidade do banco de dados).

Cada valor disponível para seleção deverá ser declarado por meio de uma *tag* interna ao *field* denominada '**item**', que têm os atributos '**value**', que será gravado no banco de dados, e '**label**', que é o rótulo exibido ao usuário:

```
<field type="Enum" column="marriage" label="State Civil | pt_BR: Estado Civil">
  <item value="_SINGL_" label="Single (a) | pt_BR: Solteiro(a)" />
  <item value="_MARRI_" label="Casado (a) | pt_BR: Casado(a)" />
  <item value="_DIVOR_" label="Divorced (a) | pt_BR: Divorciado(a)" />
</field>
```

Recomenda-se que os valores do atributo 'value' de todos os itens do campo tenham o mesmo tamanho. Desta forma, poderá ser representado no banco de dados pelo tipo '*char*' (com tamanho fixo). Para o exemplo acima ficaria:

```
ALTER TABLE titan._user ADD COLUMN marriage CHAR(7);
```

Radio

É apenas uma representação diferenciada do tipo “Enum” em formulários utilizando o botão de rádio (do inglês, *radio button*). A Figura 1.11 exemplifica esta representação para o mesmo exemplo utilizado no tipo “Enum”.

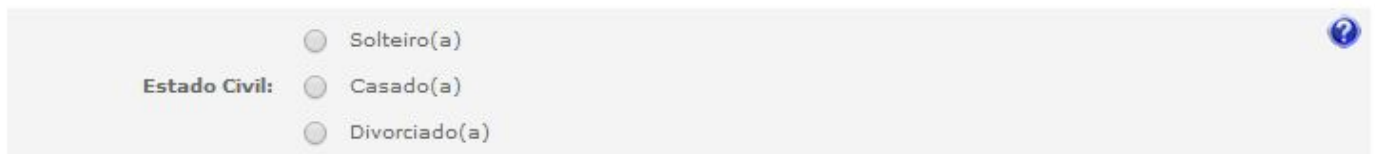
A imagem mostra um formulário web para o campo "Estado Civil". À esquerda, o rótulo "Estado Civil:" está alinhado com três opções de rádio. Cada opção consiste em um círculo cinza não selecionado seguido pelo texto: "Solteiro(a)", "Casado(a)" e "Divorciado(a)". No canto superior direito do formulário, há um ícone de interrogação azul dentro de um círculo, indicando uma ajuda ou explicação.

Figura 1.11: Campo de entrada de dados do tipo “Radio”.

CheckBox

Utilizado para possibilitar a representação de um campo de caixa de seleção (do inglês, *checkbox*), onde há diversas opções pré-definidas e o usuário escolhe quantas quiser. Repare que, assim como no tipo “Enum”, os valores aceitos são definidos pelo usuário no momento em que mapeia o campo no XML e não há relacionamento com outra entidade do banco de dados.

Figura 1.12: Campo de entrada de dados do tipo “CheckBox”.

A maior diferença deste tipo em relação ao seu pai, o tipo “Enum”, será a estrutura no banco de dados, onde a coluna deverá suportar a múltipla seleção de valores. Considere o exemplo abaixo:

```
<field type="CheckBox" column="region" label="Região Geográfica" help="Escolha uma ou mais regiões do país.">
  <item value="_CO_" label="Centro-Oeste" />
  <item value="_NE_" label="Nordeste" />
  <item value="_NO_" label="Norte" />
  <item value="_SE_" label="Sudeste" />
  <item value="_SU_" label="Sul" />
</field>
```

A representação gráfica pode ser vista na Figura 1.12 e a coluna no banco de dados ficaria:

```
ALTER TABLE tabela ADD COLUMN region CHAR(4)[];
```

Twitter

Este tipo permite que sejam postadas mensagens diretamente em uma conta do Twitter durante o salvamento de um formulário. Isto é bastante útil quando se está postando uma notícia, evento, alerta ou aviso em um portal que tenha uma conta Twitter associada.

A explicação de uso deste tipo, que é mais complexa, estará disponível no livro “**Cookbook**

Master Chef, mas você pode visualizar uma explicação detalhada na lista de discussão da ferramenta⁷.

Color

Este tipo fornece uma interface amigável utilizando paleta de cores em um campo no formulário para vínculo de uma cor à tupla. Útil para a criação de rótulos ou tipos que irão classificar ou organizar outra entidade. Na Figura 1.13 é mostrado o seu uso.



Figura 1.13: Campo de entrada de dados do tipo “Color”.

Para utilizá-lo, deve-se criar na tabela uma coluna do tipo 'char' de tamanho 6 (seis) onde será guardado a cor no padrão RGB hexadecimal (no exemplo, '2E3CFF'):

```
ALTER TABLE tabela ADD COLUMN coluna CHAR(6);
```

Slug

Este tipo permite a criação de URLs semânticas⁸ para sites que utilizem instâncias do Titan para a gestão de conteúdo. Basicamente ele permite que seja inserido no item em edição uma “string” sem espaços, acentos, letras maiúsculas ou caracteres especiais. Esta “string”, por sua vez, pode ser utilizada para referenciar o item ao invés do ID numérico (chave-privada). A maior vantagem no uso deste tipo é permitir que ferramentas de busca como o Google ranqueiem⁹ de forma melhor o conteúdo do site.

Para utilizar deve-se criar na tabela no banco de dados uma coluna VARCHAR com

⁷ <https://groups.google.com/d/topic/titan-framework/PGKMC2ryw94/discussion>

⁸ http://en.wikipedia.org/wiki/Semantic_URL

⁹ <http://pt.wikipedia.org/wiki/PageRank>

propriedades '**NOT NULL**' e '**UNIQUE**' (fundamental, pois este recurso irá substituir o ID do item na página). No XML ficará:

```
<field
  type="Slug"
  column="nome_da_coluna"
  label="URL Amigável"
  required="true"
  unique="true"
  max-length="512"
  help="Irá compor a URL do link deste item no site principal."
  base="_ID_DO_FIELD_EM_QUE_SERA_BASEADO_"
/>
```

Repare que este tipo possui um atributo específico denominado "**base**". Seu uso não é obrigatório, mas nele pode ser atribuído o ID de outro *field* que o tipo "Slug" utilizará para gerar automaticamente uma proposta de URL amigável.

TimeZone

Este é um tipo criado especialmente para aprimorar o suporte do Titan ao fuso horário do usuário. Para que as instâncias do Titan consigam mostrar a hora correta aos usuários que o utilizam, elas precisam corrigir a hora atual, obtida do servidor, com o fuso do usuário. Para obter o fuso correto, as instâncias utilizam três procedimentos.

O de maior precedência é a escolha do usuário. Para permitir que o usuário selecione seu fuso horário basta inserir no formulário de atualização de perfil (geralmente o 'modify.xml' da seção) um campo que mapeie este tipo:

```
<field type="TimeZone" label="Time Zone | pt_BR: Fuso Horário | es_ES: Huso Horario" />
```

O campo gerado será um menu *drop-down* como todos os fuso horários mundiais. Repare que não é necessário inserir o nome da coluna, que por padrão será o nome da coluna mandatória '**_timezone**' da tabela '**_user**'. Pode-se, no entanto, utilizar este campo para outros fins, que não no perfil do usuário. Neste caso pode-se alterar o atributo '**column**' de forma apropriada.

É importante salientar que, caso o usuário não tenha escolhido um fuso horário, ou seja, o

valor da coluna '**__timezone**' na tabela '**__user**' seja nulo, vazio ou caso ela não exista, o Titan tentará obter o *time zone* a partir do navegador. Caso também não consiga, por qualquer motivo, o Titan utiliza então o *time zone* definido no atributo '**timezone**' da tag '**<titan-configuration />**' do arquivo '**configure/titan.xml**' da instância.

Integer

O tipo “Integer” é também a base de diversos outros tipos do *framework*. Seu objetivo é possibilitar a entrada de números inteiros (*integers*). Desta forma, possui uma máscara no campo de entrada permitindo que apenas números inteiros sejam digitados. É representado no banco de dados pelo tipo '*integer*':

```
ALTER TABLE tabela ADD COLUMN coluna INTEGER;
```

Amount

Conforme adiantado na introdução desta seção, este tipo também é destinado a possibilitar a entrada e representação de números inteiros, porém de formatados (com pontos separando os milhares).

Boolean

O tipo “Boolean” têm por objetivo possibilitar a entrada e representação de valores lógicos ou booleanos (*booleans*). Na prática, este campo aceita dois valores: sim ou não.

Por padrão, a representação do tipo em um formulário apresenta uma caixa de seleção (*checkbox*) que pode ser marcada ou desmarcada. Entretanto, o tipo possui também um atributo específico denominado “**question**” que pode ser '*true*' ou '*false*'. Quando o atributo existir na *tag* da *field* e estiver marcado como verdadeiro (*true*), ao invés da caixa de seleção serão mostradas as opções “Sim” e “Não” para que o usuário escolha entre uma delas. Na Figura 1.14 é mostrada sua representação com este atributo ativo.

Este documento é útil?: ☐ Sim ☒ Não



Figura 1.14: Campo de entrada de dados do tipo “Boolean”.

É representado no banco de dados pelo tipo 'bit':

```
ALTER TABLE tabela ADD COLUMN coluna BIT(1) DEFAULT B'0' NOT NULL;
```

File

Este tipo é responsável pelo *upload* e visualização de arquivos nos formulários das instâncias do Titan. Ele herda de “Integer” pois o dado que é salvo dentro do banco é um número que representa o arquivo na estrutura de *upload* da instância.

Na prática, todo arquivo enviado por meio deste campo é inserido na pasta de arquivos da instância (definida no atributo '**data-path**' da tag '**archive**' do arquivo “**configure/titan.xml**”) e é referenciado por uma tupla criada na tabela mandatória '**_file**' (no *schema* padrão do Titan). Desta forma, o valor inserido por este campo será o da chave estrangeira para esta tabela. Assim, para utilizá-lo, basta criar a seguinte estrutura no banco de dados:

```
ALTER TABLE tabela ADD COLUMN coluna INTEGER;  
  
ALTER TABLE tabela ADD CONSTRAINT coluna_file_fk FOREIGN KEY (coluna)  
REFERENCES _file (_id) ON DELETE RESTRICT ON UPDATE CASCADE NOT DEFERRABLE;
```

Este campo permite também que o usuário vincule ao formulário em edição um arquivo que já tenha sido enviado em outra oportunidade. Este recurso é uma forma de evitar arquivos redundantes, poupando espaço do servidor. Para vincular estes arquivos, basta que o usuário digite uma parte do nome do arquivo no campo (pelo menos três caracteres) que aparecerá uma lista de possíveis arquivos para que ele escolha o que deseja vincular. Na Figura 1.15 é mostrada esta funcionalidade.



Figura 1.15: Utilizando um campo do tipo “File” para vincular ao formulário em edição um arquivo previamente enviado à instância.

Este campo possui alguns atributos específicos. São eles:

- **owner-only:** Quando este atributo existe e têm valor *'true'*, os arquivos previamente listados para associação são apenas aqueles que foram enviados pelo usuário;
- **show-details:** Por padrão este atributo possui valor *'true'*. Quando ativo, irá exibir ao lado do arquivo associado as seguintes informações: nome do arquivo, tamanho (em *bytes*), *mime type* e descrição (se houver); e
- **resolution:** Quando for uma imagem, o valor deste atributo determinará as dimensões do *thumbnail*¹⁰ exibido.

Além destes atributos, este tipo permite que seja declarada uma lista de *tags* “**mime-type**” internas. Trata-se dos tipos de arquivos (identificados por seu “MIME Type”¹¹) que o campo aceitará. Estes tipos devem ter sido previamente declarados no arquivo de configuração “**configure/archive.xml**”. Caso não seja declarada nenhuma restrição no *field*, ou seja, não haja nenhuma *tag* “**mime-type**” interna, o campo aceitará todos os tipos de arquivo habilitados para a instância.

Por exemplo, para declarar um campo que aceita apenas imagens, teríamos:

```
<field
  type="File"
  column="photo"
  label="Photo | pt_BR: Foto"
```

¹⁰ <http://pt.wikipedia.org/wiki/Thumbnail>

¹¹ http://en.wikipedia.org/wiki/Internet_media_type


```
tip="200 x 200 pixels"
owner-only="true"
show-details="false"
resolution="200"
help="Resolução recomendada de 200 pixels de largura por 200 pixels de altura.">
<mime-type>image/jpeg</mime-type>
<mime-type>image/gif</mime-type>
<mime-type>image/pjpeg</mime-type>
<mime-type>image/png</mime-type>
<mime-type>image/x-bitmap</mime-type>
<mime-type>image/photoshop</mime-type>
<mime-type>image/bmp</mime-type>
</field>
```

Double

O tipo “Double” objetiva possibilitar a entrada de números reais (ponto flutuante). Possui um atributo denominado “**precision**”, onde é declarado o número de casas decimais com que o campo irá trabalhar (por padrão é dois). Possui uma máscara no campo de entrada permitindo que apenas números reais (com a quantidade de casas decimais definida) sejam digitados. É representado no banco de dados pelo tipo *'double precision'*:

```
ALTER TABLE tabela ADD COLUMN coluna DOUBLE PRECISION DEFAULT 0.0 NOT NULL;
```

Money

O tipo “Money” objetiva possibilitar a entrada e representação de valores monetários. Funciona como o “Double”, mas possui um atributo específico denominado “**currency**”, onde o usuário deve colocar a moeda (p.e., “R\$”).

Select

O tipo “Select” e suas derivações são os principais responsáveis pela representação de relacionamentos na instância. O tipo “Select”, especificamente, possibilita a representação e associação de valores em relacionamentos do tipo “**um para vários**” (“1 x n”). Considere a Figura 1.16, nela é mostrada duas tabelas relacionadas no banco de dados. A chave estrangeira da tabela “sector” é a coluna “type”, que referencia a chave primária da tabela “type” denominada “id”.

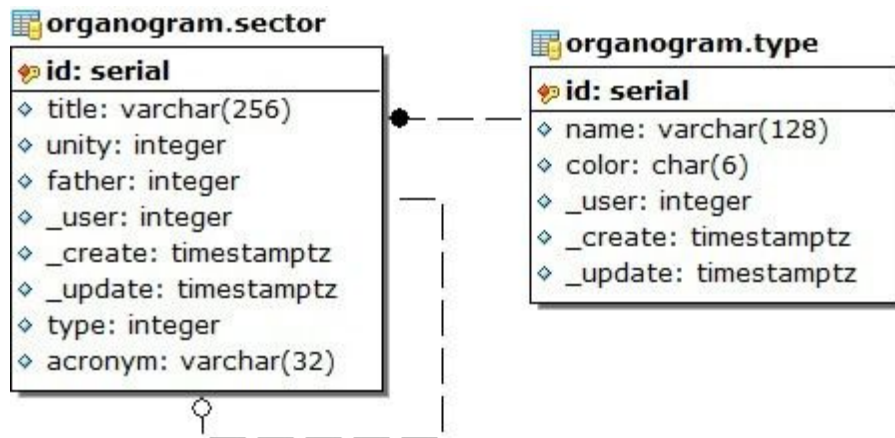


Figura 1.16: Relacionamento “1 x n” que será representado com o uso do tipo “Select”.

Este tipo possui os seguintes atributos específicos para possibilitar o mapeamento do relacionamento:

- **link-table:** A tabela que será relacionada;
- **link-column:** A chave primária da tabela relacionada;
- **link-view:** As colunas da tabela relacionada a serem exibidas. Caso queira exibir mais de uma coluna, pode-se formatar uma máscara utilizando os nomes das colunas entre colchetes. Por exemplo, configurar o atributo de um *field* que relaciona a tabela mandatória “_user” com o valor “[_name] ([_email])” exibiria algo como “Camilo Carromeu (camilo@carromeu.com)”;
- **link-color:** Caso a entidade relacionada possua um *field* do tipo “Color”, pode-se referenciar o nome da coluna dele neste atributo para que, na listagem de tuplas, os valores referenciados sejam exibidos com cor de fundo. O uso deste recurso é mostrado na Figura 1.17;

Unidade	Nome	Sigla	Tipo	Vínculo	Último Autor	
CNPGC	Chefia Adjunta de Administração	CHADM	Chefia	Chefia Geral	Camilo Carromeu	
CNPGC	Chefia Adjunta de Pesquisa e Desenvolvimento	CHPD	Chefia	Chefia Geral	Camilo Carromeu	
CNPGC	Chefia Adjunta de Transferência de Tecnologia	CHTT	Chefia	Chefia Geral	Camilo Carromeu	
CNPGC	Núcleo de Comunicação Organizacional	NCO	Núcleo	Chefia Geral	Camilo Carromeu	
CNPGC	Núcleo de Desenvolvimento Institucional	NDI	Núcleo	Chefia Geral	Camilo Carromeu	
CNPGC	Núcleo de Tecnologia da Informação	NTI	Núcleo	Chefia Geral	Camilo Carromeu	
CNPGC	Secretaria	SEC	Setor	Chefia Geral	Camilo Carromeu	
CNPGC	Laboratórios	LAB	Setor	Chefia Adjunta de Pesquisa e Desenvolvimento	Camilo Carromeu	

Figura 1.17: Listagem de itens da tabela “sector”, onde o tipo “Select” foi utilizado para representar os “tipos de setores” (coluna “Tipo” na imagem). Neste caso o atributo “link-color” do *field* aponta para a coluna “color” da tabela relacionada “type”.

- **link-api:** Este atributo será utilizado em instâncias que possuem camadas de serviços ativas (REST-Like API) e utilizem a abordagem de “desambiguação”. Este recurso é explicado detalhadamente no “**Cookbook Master Chef**”; e
- **search:** Este atributo permite que seja associado um arquivo XML com a sintaxe de engine de listagem ao *field*. Com isso, passa a ser possível efetuar busca no campo, como se faz em uma ação de listagem convencional.

No exemplo, o código XML para a representação do campo seria:

```
<field
  type="Select"
  column="type"
  label="Type | pt_BR: Tipo"
  required="true"
  link-table="organogram.type"
  link-column="id"
  link-view="name"
  help="Select the type of sector. | pt_BR: Selecione o tipo do setor."
/>
```

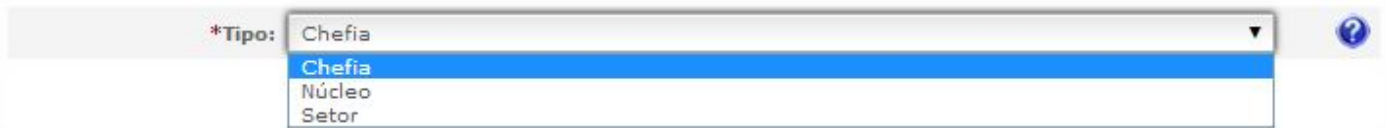


Figura 17: Campo do tipo “Select”.

A renderização é mostrada na Figura 17. Se, por exemplo, quiséssemos utilizar o atributo “search”, precisaríamos criar um novo arquivo de marcação para configurar a visualização da busca de itens. Para o exemplo que estamos utilizando, este arquivo poderia ser chamado de “search.xml” e ser colocado na pasta da seção com o seguinte conteúdo:

```
<?xml version="1.0" encoding="UTF-8"?>
<view table="organogram.type" primary="id" paginate="15">
  <field type="Phrase" column="name" label="Nome" id="_TITLE_" />
  <field type="Select" column="_user" label="Last Author | pt_BR: Último Autor"
  link-table="_user" link-column="_id" link-view="_name" />
  <order id="_TITLE_" invert="false" />
</view>
<search table="organogram.type">
  <field type="Phrase" column="name" label="Nome" />
</search>
```

Neste caso, a renderização do campo contaria com um ícone em formato de lupa, que permite ativar a busca, conforme mostrado na Figura 1.18.



Figura 1.18: Campo do tipo “Select” com o atributo “search” sendo utilizado.

Cascade

O tipo “Cascade” é derivado de “Select”. Objetiva vincular tuplas de uma tabela de forma “recursiva”, ou seja, representar entidades que possuem chave estrangeira para ela mesma, caracterizando relações de 'tuplas-pais' e 'tuplas-filhas'.

Terá um atributo denominado “**link-father**”, que contém o nome da coluna na tabela relacionada que aponta para a 'tupla-pai'.

Para exemplificar, considere a tabela “organogram.sector” mostrada na Figura 15. Sua coluna “father” é uma chave estrangeira para a chave primária (coluna “id”) da própria tabela. O DDL desta entidade é o seguinte:

```
CREATE TABLE organogram.sector (  
  id SERIAL,  
  title VARCHAR(256) NOT NULL,  
  father INTEGER,  
  _user INTEGER NOT NULL,  
  _create TIMESTAMP WITH TIME ZONE DEFAULT now() NOT NULL,  
  _update TIMESTAMP WITH TIME ZONE DEFAULT now() NOT NULL,  
  acronym VARCHAR(32) NOT NULL,  
  CONSTRAINT sector_pkey PRIMARY KEY(id),  
  CONSTRAINT sector_father_fk FOREIGN KEY (father)  
    REFERENCES organogram.sector(id)  
    ON DELETE RESTRICT  
    ON UPDATE CASCADE  
    NOT DEFERRABLE,  
  CONSTRAINT sector_user_fk FOREIGN KEY (_user)  
    REFERENCES titan._user(_id)  
    ON DELETE RESTRICT  
    ON UPDATE CASCADE  
    NOT DEFERRABLE  
);
```

Assim, um “setor” pode estar (ou não) vinculado a um “setor-pai”. Em tabelas com tuplas que irão se relacionar aos setores é inserida uma coluna com uma chave estrangeira para esta tabela:

```
ALTER TABLE tabela ADD COLUMN coluna INTEGER;  
  
ALTER TABLE tabela ADD CONSTRAINT coluna_sector_fk FOREIGN KEY (coluna)  
REFERENCES organogram.sector (id) ON DELETE RESTRICT ON UPDATE CASCADE NOT DEFERRABLE;
```

Repare que, ao utilizar esta abordagem para representar dados organizados em múltiplos níveis, a chave estrangeira sempre será respeitada independente do nível que está sendo referenciado. Além disso, esta arquitetura permite uma quantidade infinita de níveis na árvore.

Para mapear o tipo nos XMLs:

```
<field
  type="Cascade"
  column="coluna"
  label="Setor"
  link-table="organogram.sector"
  link-column="id"
  link-view="title"
  link-father="father"
/>
```

Na Figura 1.19 é mostrado como é visualizado o campo para entrada de dados utilizando este tipo.

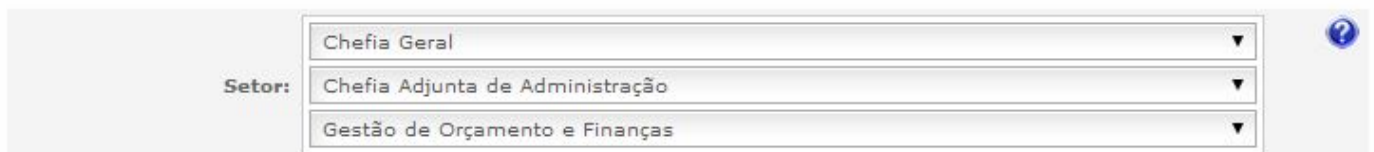


Figura 1.19: Campo do tipo “Cascade”.

City

O tipo “City” é derivado de “Select”, com a diferença de que já é configurado para relacionar a tabela de cidades do Titan (tabela mandatória “_city” do *schema* padrão). Também possui um atributo denominado “uf”, onde deve ser especificado qual o estado (unidade federativa) do Brasil do qual serão as cidades listadas. Assim, para trabalhar com este campo, deve-se utilizar este atributo ou utilizá-lo em conjunto com o tipo “State” (abaixo), ou seja, é fundamental que o campo receba o estado ao qual pertencem as cidades exibidas.

State

O tipo “State”, derivado de “Select”, já é pré-configurado para relacionar a tabela de estados

do Titan (tabela mandatória “_state” do *schema* padrão). Pode ser utilizado sozinho ou em conjunto com o tipo “City” (acima). Para este último caso, ele possui um atributo específico denominado “city-id”, que é o identificador (atributo “id”) do *field* do tipo “City” que será carregado com as cidades pertencentes ao estado selecionado neste campo.

Uma declaração típica destes campos, utilizados em conjunto, no arquivo XML da seção seria:

```
<field type="State" column="state" label="State | pt_BR: Estado" value="MS" city-id="_CITY_" />
<field type="City" column="city" label="City | pt_BR: Município" uf="MS" id="_CITY_" />
```

Multiply

Este tipo é muito semelhante ao “Select” mas, ao invés de representar uma relação “um para vários” (“1 x n”), possibilitará a representação de uma relação “**vários para vários**” (“n x n”). Para utilizá-lo será necessário criar no banco de dados uma tabela de relacionamento, ou seja, a tabela que irá fazer a relação “n x n” das tuplas de duas outras tabelas. Para abrigar a declaração desta tabela, este tipo possui os seguintes atributos específicos

- **relation:** O nome da tabela de relacionamento;
- **primary:** É o nome da chave primária da tabela mapeada pelo XML e deve ser preenchido quando o *field* estiver sendo utilizado em uma *tag* “search”, ou seja, em um formulário de busca;
- **check-box:** Aceita 'true' ou 'false' e determina se a exibição do campo deve ser feita no formato de caixas de marcação (*checkbox*) ao invés de um menu *drop down*; e
- **relation-link:** Por padrão o Titan irá considerar que os nomes das colunas na tabela de relacionamento serão os mesmos das tabelas que eles relacionam. Isto, entretanto, nem sempre será verdade. Este atributo possibilita especificar o nome da coluna na tabela de relacionamento.

Para exemplificar, vamos supor que haja uma tabela 'filme' e outra tabela 'gênero'. Um filme pode estar associado a diversos gêneros e um gênero pode estar classificando diversos filmes. Neste caso, as tabelas a serem criadas serão:

```
CREATE TABLE store.movie (
  id SERIAL,
  title VARCHAR(256) NOT NULL,
  description TEXT,
  _user INTEGER NOT NULL,
  _create TIMESTAMP WITH TIME ZONE DEFAULT now() NOT NULL,
  _update TIMESTAMP WITH TIME ZONE DEFAULT now() NOT NULL,
  CONSTRAINT movie_pkey PRIMARY KEY(id),
  CONSTRAINT movie_user_fk FOREIGN KEY (_user) REFERENCES titan._user(_id)
  ON DELETE RESTRICT ON UPDATE CASCADE NOT DEFERRABLE
);

CREATE TABLE store.genre (
  id SERIAL,
  name VARCHAR(256) NOT NULL,
  _user INTEGER NOT NULL,
  _create TIMESTAMP WITH TIME ZONE DEFAULT now() NOT NULL,
  _update TIMESTAMP WITH TIME ZONE DEFAULT now() NOT NULL,
  CONSTRAINT genre_pkey PRIMARY KEY(id),
  CONSTRAINT genre_user_fk FOREIGN KEY (_user) REFERENCES titan._user(_id)
  ON DELETE RESTRICT ON UPDATE CASCADE NOT DEFERRABLE
);

CREATE TABLE store.movie_genre (
  movie INTEGER,
  genre INTEGER,
  CONSTRAINT movie_genre_pkey PRIMARY KEY(movie, genre),
  CONSTRAINT movie_fk FOREIGN KEY (movie) REFERENCES store.movie(id)
  ON DELETE CASCADE ON UPDATE CASCADE NOT DEFERRABLE,
  CONSTRAINT genre_fk FOREIGN KEY (genre) REFERENCES store.genre(id)
  ON DELETE RESTRICT ON UPDATE CASCADE NOT DEFERRABLE
);
```

O resultado da execução dos SQLs acima é a estrutura mostrada na Figura 1.20.

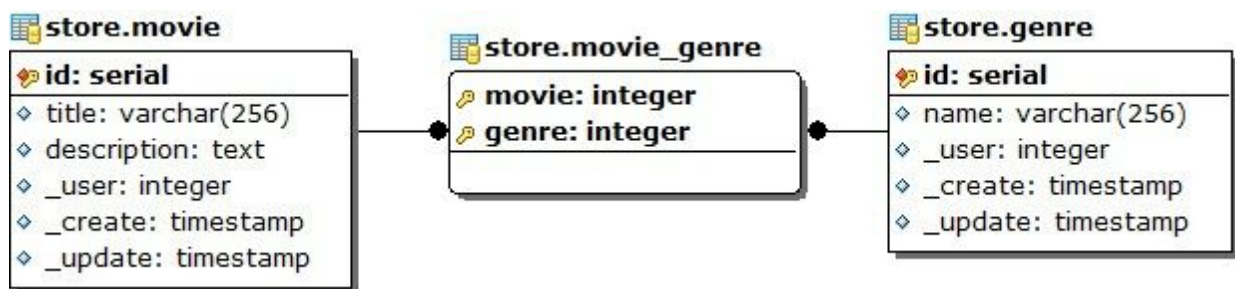


Figura 1.20: Relacionamento “ $n \times n$ ” que será representado com o uso do tipo “Multiply”.

No exemplo, vamos considerar que o 'gênero' é uma caracterização do 'filme'. Assim, quando um filme é cadastrado são vinculados gêneros a ele. Haverá, portanto, uma diferença sutil nas chaves estrangeiras da tabela “movie_genre”. Repare que quando um gênero é apagado, há uma restrição caso este gênero esteja vinculado a algum filme. Entretanto, no caso de um filme ser

apagado, todas as ligações deste filme com gêneros são apagados em cascata.

Da mesma forma, a declaração do campo do tipo “Multiply” será feita no formulário de edição do filme:

```
<?xml version="1.0" encoding="UTF-8"?>
<form table="store.movie" primary="id">
...
<field type="Multiply" column="movie_genre" label="Gêneros" link-table="genre" link-column="id"
link-view="name" relation="store.movie_genre" />
...
</form>
```

Collection

O tipo “Collection” objetiva permitir a criação de tuplas *in loco* em tabelas relacionadas à entidade em que o campo deste tipo é inserido, ou seja, permite criar, apagar e exibir elementos vinculados ao formulário em uma relação “**um para vários**” (“1 x n”). Desta forma, este tipo irá renderizar para o usuário uma 'sub-lista' dos itens relacionados e um 'sub-formulário' para a criação de novos itens. A definição destes elementos deverá ser feita em um XML próprio, que será referenciado no field por meio do atributo específico denominado “**xml-path**”.

Por exemplo, suponha a seguinte estrutura de tabelas:

```
CREATE TABLE public.book (
  id SERIAL,
  name VARCHAR(128) NOT NULL,
  _user INTEGER NOT NULL,
  _create TIMESTAMP WITH TIME ZONE DEFAULT now() NOT NULL,
  _update TIMESTAMP WITH TIME ZONE DEFAULT now() NOT NULL,
  CONSTRAINT book_pkey PRIMARY KEY(id),
  CONSTRAINT book_user_fk FOREIGN KEY (_user)
    REFERENCES titan._user(_id)
    ON DELETE RESTRICT
    ON UPDATE CASCADE
    NOT DEFERRABLE
) WITHOUT OIDS;

CREATE TABLE public.author (
  id SERIAL,
  book INTEGER NOT NULL,
  first_name VARCHAR(64) NOT NULL,
  last_name VARCHAR(64) NOT NULL,
  _create TIMESTAMP WITH TIME ZONE DEFAULT now() NOT NULL,
  CONSTRAINT author_pkey PRIMARY KEY(id),
  CONSTRAINT author_book_fk FOREIGN KEY (book)
    REFERENCES public.book(id)
    ON DELETE CASCADE
```

```
ON UPDATE CASCADE
NOT DEFERRABLE
) WITHOUT OIDS;
```

Repare que na tabela 'author' há uma chave estrangeira para a tabela 'book', formando uma relação "1 x n". O "Collection" permitirá que o usuário, no formulário de edição de livros, crie autores que são automaticamente vinculados ao livro editado. O tipo conta, portanto, com um formulário que permite a inserção de autores e uma listagem dos autores inseridos (que permite deletá-los). Assim, é necessário prover a este tipo o modelo de dados deste formulário e desta listagem por meio de um XML próprio, criado na pasta da seção:

```
<?xml version="1.0" encoding="UTF-8"?>
<view table="public.author" primary="id">
  <field type="Phrase" column="first_name" label="First Name | pt_BR: Primeiro Nome" />
  <field type="Phrase" column="last_name" label="Last Name | pt_BR: Último Nome" />
</view>
<form table="public.author" primary="id">
  <field type="Phrase" column="first_name" label="First Name | pt_BR: Primeiro Nome"
    max-length="64" required="true" />
  <field type="Phrase" column="last_name" label="Last Name | pt_BR: Último Nome"
    max-length="64" required="true" />
</form>
```

Vamos supor que o XML criado acima esteja na pasta da seção de livros com o nome 'author.xml'. Agora, basta inserir o campo no formulário de livros:

```
<field type="Collection" id="_AUTHOR_" column="book" xml-path="author.xml" />
```

A renderização do campo do exemplo pode ser visualizada na Figura 1.21.

Autores

Inserir Novo Item

*Primeiro Nome: 64

*Último Nome: 64

Adicionar Item Fechar

Primeiro Nome	Último Nome	
Olavo de	Carvalho	
Rodrigo	Constantino	
Leandro	Narloch	

Figura 1.21: Campo do tipo “Collection”.

Algumas informações importantes sobre o tipo “Collection”:

- Somente é possível editar um campo do tipo “Collection” em ações de edição, ou seja, a tupla "mãe" já precisa estar criada para que o “Collection” saiba a qual entidade ele deve relacionar o item criado;
- Como, na prática, são formulários independentes, os campos “Collection” sempre ficarão no final da página de edição, independente do local em que você os colocou no XML. Em ações de visualização a posição escolhida é respeitada; e
- Sempre utilize os mesmos campos na *tag* 'view' e na *tag* 'form' do XML do “Collection”. O JavaScript que cria uma nova linha na visualização lida apenas com os campos declarados na *tag* 'form', ou seja, se houverem números distintos de colunas na 'view' e na 'form' a visualização ficará prejudicada.

Document

O tipo “Document” possui uma complexa estrutura que adiciona às instâncias do Titan a capacidade de gerar e validar documentos PDF. A explicação de uso deste tipo estará disponível no livro **“Cookbook Master Chef”**, mas você pode visualizar uma explicação detalhada na lista de

discussão da ferramenta¹².

Date

O tipo “Date” é utilizado para possibilitar a entrada e representação de datas. Possui os seguintes atributos específicos:

- **first-year:** Configura o primeiro ano que poderá ser selecionado no campo. Caso não seja especificado, o padrão será 30 anos antes do ano atual. Aceita a palavra-chave “[now]” para determinar que o primeiro ano é o atual;
- **last-year:** Configura o último ano que poderá ser selecionado no campo. Caso não seja especificado, o padrão será 30 anos após o ano atual. Aceita a palavra-chave “[now]” para determinar que o último ano é o atual;
- **show-time:** Está disponível para possibilitar a representação, em um único campo, de valores *timestamp* com data e hora. Aceita os valores 'true' e 'false'. Caso seja especificado com 'true', em ações de visualização será mostrado com a hora ao lado da data. É útil, principalmente, para representar valores gerados automaticamente no banco de dados, como em colunas que recebem, por padrão, o valor do método “now ()”; e
- **show-age:** Aceita os valores 'true' e 'false'. Caso seja especificado com 'true' irá mostrar em ações de visualização, ao lado da data, o tempo em anos decorridos até a data atual.

É representado no banco de dados pelo tipo '*timestamp*' (que pode ser com ou sem *timezone*, dependendo do seu requisito):

```
ALTER TABLE tabela ADD COLUMN coluna TIMESTAMP WITHOUT TIME ZONE;
```

Abaixo é exibido um exemplo de um campo para preenchimento da “Data de Nascimento” e na Figura 1.22 é mostrada sua renderização.

```
<field
  type="Date"
  column="birth_date"
  label="Birth Date | pt_BR: Data de Nascimento"
  first-year="1900"
  last-year="[now]"
```

¹² https://groups.google.com/d/topic/titan-framework/LM3-_frfiI8/discussion

```
show-age="true"
```

```
/>
```

Data de Nascimento: 19 ▾ Julho ▾ 2014 ▾

JULHO 2014

HOJE						
<<	<				>	>>
dom	seg	ter	qua	qui	sex	sab
		1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30	31		

Figura 1.22: Campo do tipo “Date”.

Time

O tipo “Time” é utilizado para possibilitar a entrada e representação de hora. É representado no banco de dados pelo tipo 'timestamp' (que pode ser com ou sem *timezone*, dependendo do seu requisito):

```
ALTER TABLE tabela ADD COLUMN coluna TIMESTAMP WITH TIME ZONE DEFAULT now() NOT NULL;
```

1.6 Boas Práticas de Programação

Boas práticas de programação são um conjunto de regras informais que visam melhorar a qualidade das aplicações e simplificar a sua manutenção, reduzindo significativamente a probabilidade de erros em suas aplicações, além de facilitar o entendimento de novos códigos de forma rápida e completa. Esta seção apresentará as convenções recomendadas para Titan Framework, que está sendo adequado às normas propostas em um modelo de convenções do PHP chamado PSR-o (*Proposing a Standards Recommendation*). O objetivo de abordar este tema

agora é garantir que as diretrizes elencadas sejam seguidas desde as primeiras instâncias desenvolvidas pelo leitor.

1.6.1 Convenções para o Banco de Dados

- O nome de todas as entidades do banco de dados devem estar em **inglês**;
- O nome de todas as entidades do banco de dados devem estar no **singular**;
- O nome de todas as entidades do banco de dados devem estar em **caixa baixa** (*lower case*);
- Entidades “**mandatórias**”, ou seja, cujo a existência, o nome ou o comportamento não possam ser parametrizados pelos arquivos de marcação da instância, devem ser **precedidas por underscore** (por exemplo, o componente nativo “**global.generic**” necessita que as tabelas que representam o modelo de dados de suas seções tenham as colunas “**_user**”, “**_create**” e “**_update**”); e
- Caso o nome da entidade seja composto de duas ou mais palavras, elas devem estar separadas por **underscore** (p.e., “my_table”).

1.6.2 Convenções de Código

- Os nomes de variáveis, funções, classes e demaís entidades devem estar em **inglês**;
- Os nomes de variáveis locais, atributos e funções devem estar em **Camel Case** (a primeira palavra do nome é iniciada por letra minúscula, a próxima palavra é iniciada com maiúscula e todas são unidas sem qualquer caracter), tal como “\$myFirstVar” ou “myFunction ()”;
- Caso seja um atributo de classe privado, o nome deve ser precedido com um **underscore** (p.e., “\$_myPrivateAttribute”);
- Os nomes das classes devem estar em **Pascal Case** (todas as palavras são iniciadas maiúsculas e são unidas sem espaço), tal como “MyClass”;
- Os nomes das variáveis globais devem estar em caixa alta (*upper case*), tal como

“\$_MY_VAR”;

- O nome das constantes devem estar em **caixa alta** (*upper case*), tal como “MY_CONST”;
- Utilize **Orientação a Objetos** e suas boas práticas;
- Busque manter funções pertinentes dentro das classes (mesmo que sejam métodos estáticos);
- Todas as *tags* e atributos em XML devem possuir apenas letras minúsculas (*lower case*);
- Sempre quebre a linha após ponto-e-vírgula, antes de um operador ou antes de abrir chaves (delimitador de escopo);
- Sempre utilize **tabulação** para a **identação** (jamais utilize o “espaço”). Se necessário, configure na IDE de desenvolvimento pois algumas, por padrão, utilizam o caracter de espaço;
- Sempre feche parênteses e chaves na mesma coluna ou na mesma linha em que foram abertos;
- Busque utilizar linhas em branco entre funções e métodos, entre classes e entre expressões com variáveis distintas;
- Em todos os arquivos deve-se utilizar o padrão de cabeçalhos e comentários especificado pelo PHPDoc¹³;
- Sempre que aplicável, utilize *Design Patterns*; e
- Jamais utilize *deprecated features*.

Há um *profile* para a **IDE Eclipse** com as definições acima já configuradas disponível no repositório do Titan¹⁴. Uma vez que ele esteja importado na IDE de desenvolvimento, é possível utilizar a combinação de teclas “CTRL + SHIFT + F” para formatar automaticamente o código segundo a convenção sugerida. Para ativar o profile deve-se acessar a opção "Window >

¹³ <http://www.phpdoc.org/>

¹⁴ https://svn.cnpge.embrapa.br/titan/document/PLEASE_Profile.xml

Preferences" do Eclipse e, em seguida, clicar no botão "Import", como mostrado na Figura 1.23.

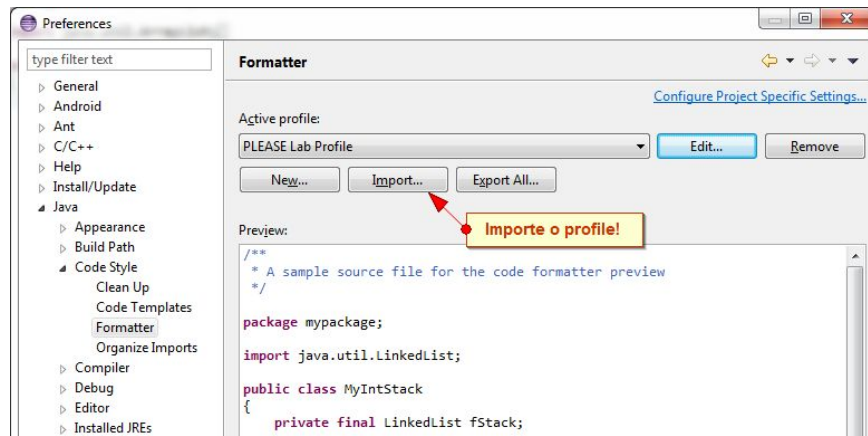


Figura 1.23: Importação do *profile* com a convenção de código para a IDE Eclipse.

II. Criando a Primeira Aplicação

Neste capítulo iremos colocar a “mão na massa”, desenvolvendo uma aplicação funcional com o uso do Titan. Caso queira vislumbrar o funcionamento de instâncias sem ter que desenvolver uma, há exemplos completos no diretório “**sample**” na raiz do repositório do *framework*¹⁵.

2.1 Ambiente de Desenvolvimento

O Titan utiliza o **Vagrant**¹⁶ para padronizar o ambiente de desenvolvimento. Há uma “*base-box*” já configurada que pode ser facilmente obtida e instalada. Para utilizar, primeiramente instale o **VirtualBox** e o **VirtualBox Extension Pack**:

<https://www.virtualbox.org/wiki/Downloads>

Em seguida, faça download e instale o **Vagrant**:

<http://www.vagrantup.com/downloads.html>

Feito isso, a *base-box* para o Titan deve ser carregada em seu sistema para ser utilizada nos diversos projetos implementados com o *framework*. Basta abrir o *command* (no Windows, '**cmd.exe**') e dar o comando:

```
C:\> vagrant box add TitanBox 'http://titan.cnpqc.embrapa.br/Titan.box'
```

A *box* têm cerca de 550 MB, então pode demorar dependendo da velocidade de sua conexão. Outra forma de carregar a *box* é fazendo o download pelo seu gerenciador preferido e passar, no último parâmetro, o caminho para o arquivo '**Titan.box**' em seu sistema de arquivos.

2.2 Instanciação da Aplicação

No contexto da orientação a objetos, instância significa a concretização de uma classe. Em

¹⁵ <https://svn.cnpqc.embrapa.br/titan/sample>

¹⁶ <http://docs.vagrantup.com/v2/why-vagrant/index.html>

termos intuitivos, uma classe é vista como um “molde” que gera instâncias de um certo tipo, já objeto é algo que existe fisicamente criado a partir deste molde. No Titan, a instanciação é a concretização, por meio de um conjunto de arquivos que referenciam o *core*, de uma nova aplicação Web.

Para a criação da primeira instância no Titan, usaremos como estudo de caso o desenvolvimento de um sistema para bibliotecas. Para isso, será realizado o *export* de uma instância básica, ou seja, uma instância funcional, porém com os artefatos mínimos que normalmente estão presentes em todas as instâncias do Titan. É a partir da evolução da instância básica que será desenvolvida toda a aplicação final.

Assim, para começar, dê um *export* da instância básica do Titan para criar seu projeto. O comando abaixo sugere que sua pasta de projetos está em “C:\projetos” e que você tem um cliente Subversion de linha de comando instalado em seu computador. Você pode efetuar o mesmo comando utilizando a opção “TortoiseSVN » Export” do menu de contexto que aparece ao clicar o segundo botão do mouse sobre a pasta “C:\projetos\library”, desde que tenha o TortoiseSVN¹⁷ instalado em seu Windows.

```
C:\> svn export https://svn.cnpqc.embrapa.br/titan/instance C:\projetos\library
```

Ao final, a estrutura de pastas da instância básica criada dentro da sua pasta “C:\projetos\library” deverá estar conforme é mostrado na Figura 2.1. Repare que teremos três diretórios na raiz:

- **app:** Diretório com os arquivos da aplicação (scripts PHP e XMLs de parametrização);
- **box:** Diretório com o “Vagrantfile”, ou seja, por meio do qual será iniciado o ambiente de desenvolvimento; e
- **db:** Diretório com o arquivo “last.sql”, que contém um *dump* com a estrutura (entidades) a ser criada no banco de dados para o bom funcionamento da instância básica.

¹⁷ <http://tortoisesvn.net/downloads.html>

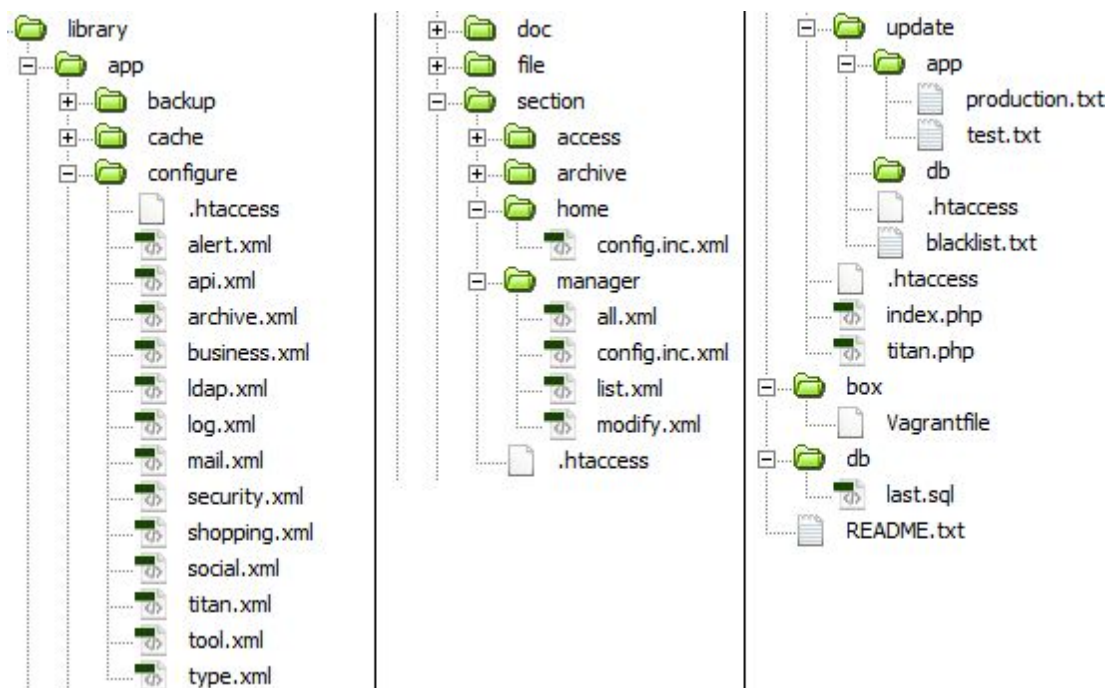


Figura 2.2: Estrutura de diretórios criada após a exportação da instância básica do Titan.

O primeiro destes diretórios a qual devemos nos atentar é o “**box**”. Abra o *command* (“cmd.exe”), vá até este diretório e dê o comando para que a TitanBox seja inicializada:

```
C:\projetos\library\box> vagrant up
```

Com isso, será criada e inicializada em *background* uma máquina virtual (do inglês, *virtual machine*) rodando o sistema operacional Linux Debian Wheezy 64 bits com Apache 2.2, PHP 5.4 e PostgreSQL 9.1. Ela possui todos os recursos recomendados para o Titan já instalados e já têm o *core* embutido (no diretório “/var/www/titan” do Debian).

Quando a máquina virtual é inicializada o Vagrant mapeia os diretórios do seu projeto para diretórios do Debian da seguinte forma:

- **app** » /var/www/app
- **box** » /vagrant
- **db** » /var/lib/postgresql/db

Além disso, por padrão as seguintes portas serão mapeadas:

- A porta **80 (Apache)** da *guest* para a **8090** do *host*;
- A porta **5432 (PostgreSQL)** da *guest* para a **5431** do *host*; e
- A porta **22 (SSH)** da *guest* para a **2222** do *host*.

Assim, a máquina virtual ficará acessível por SSH. Como a porta do SSH está mapeada para o *host*, basta abrir seu cliente SSH e conectar em “**localhost**” ou “**127.0.0.1**” na porta “**2222**”. Os usuários disponíveis são:

- login “**root**” com senha “**vagrant**”; e
- login “**vagrant**” com senha “**vagrant**”.

Caso tenha um cliente SSH de linha de comando instalado e acessível pelo *path* do Windows, outra forma de acessar o *shell* da máquina virtual é simplesmente executar o seguinte comando:

```
C:\projetos\library\box> vagrant ssh
```

Pronto, com isso todos os arquivos colocados na pasta “**app**” do seu projeto serão sincronizados na VM e estarão disponíveis pelo navegador em “**http://localhost:8090**”. A mesma lógica é aplicada ao PostgreSQL (“**localhost**” na porta “**5431**”).

O banco de dados da instância deve agora ser importado para o ambiente:

```
C:\projetos\library\box> vagrant ssh
vagrant@titan:~$ sudo su - postgres
postgres@titan:~$ createdb -E utf8 -O titan library
postgres@titan:~$ psql -d library -U titan < db/last.sql
```

Repare que já existe um usuário denominado “**titan**” (com senha “**titan**”) criado no PostgreSQL.

Para configurar seu *database* na instância, volte sua atenção para o Windows novamente. Abra o arquivo “**C:\projetos\library\app\configure\titan.xml**” na sua IDE predileta. Este arquivo, conforme já visto, armazena as principais configurações da instância: o nome da aplicação, o nome do autor, as *hashs* de segurança e criptografia, os dados do *database*, entre

outros. O arquivo deve ser editado considerando o ambiente da máquina virtual. No exemplo abaixo foram rachuradas as linhas que precisaram ser modificadas para nossa instância “*library*”.

```
<?xml version="1.0" encoding="UTF-8"?>
<titan-configuration
  name="Minha Biblioteca"
  description="Minha Primeira Instância do Titan"
  url="http://localhost:8090/"
  e-mail="seu@e-mail.com"
  login-url="http://localhost:8090/titan.php?target=login"
  core-path=" ../titan/"
  cache-path="cache/"
  doc-path="doc/"
  debug-mode="true"
  session="minha_hash_secreta_para_a_secao"
  language="pt_BR, en_US"
  use-chat="false"
  all-sections="true"
  author="&copy; 2014 &curren; Seu Nome ou da Sua Instituição/Empresa">

  <database
    sgbd="PostgreSQL"
    host="localhost"
    name="library"
    user="titan"
    port="5432"
    password=""
    schema="titan"
  />

  <security
    xml-path="configure/security.xml"
    hash="minha_hash_secreta_para_a_criptografia"
    timeout="1800"
  />

  <search
    hash="minha_hash_secreta_para_cookies"
    timeout="15552000"
  />

  <archive
    xml-path="configure/archive.xml"
    data-path="file/"
  />

  <business-layer
    xml-path="configure/business.xml"
  />

  <skin
    logo=""
  />

  <mail
    xml-path="configure/mail.xml"
  />
```

```
<version-control
  schema="version"
/>

<log
  db-path="cache/log.db"
  xml-path="configure/log.xml"
/>

<!--
<type
  xml-path="configure/type.xml"
  use-legacy="true"
/>

<tool
  xml-path="configure/tool.xml"
/>

<social
  xml-path="configure/social.xml"
/>

<api
  xml-path="configure/api.xml"
/>

<shopping
  currency="BRL"
  xml-path="configure/shopping.xml"
/>
-->

<lucene
  index-path="cache/lucene"
/>

<schedule
  hash=""
/>

<alert
  xml-path="configure/alert.xml"
/>

<friendly-url
  change-password="password"
  disable-alerts="alert"
  rss="rss"
/>

<backup
  path="backup/"
  validity="86400"
  timeout="86400"
/>
</titan-configuration>
```

Note as *tags* utilizadas neste arquivo. A primeira, “**titan-configuration**”, descreve em seus atributos a configuração da sua instância no Titan, como o **nome do autor**, da **aplicação**, e alguns caminhos como o “**core-path**” (o caminho para o núcleo imutável do Titan), o “**cache-path**” (a pasta onde o Titan irá cachear arquivos visando otimização e desempenho) e “**doc-path**” (onde poderá ser sobreescrito conteúdo do manual do usuário). Esta *tag* também contém uma série de configurações adicionais, tal como de banco de dados, segurança, e-mails, alertas, entre outros.

A *tag* “**database**” configura justamente o banco de dados usado na aplicação, contendo atributos como o nome do banco de dados, o usuário, o esquema principal, entre outros. Você precisará alterar apenas o atributo “**name**” desta *tag*. Não coloque senha, pois assim o Titan utilizará **sockets Unix** para conectar ao PostgreSQL, o que possibilitará o uso da funcionalidade *backup on demand*¹⁸ do *framework*.

Após efetuar as alterações indicadas (rachuradas), salve o arquivo e acesse a sua instância pelo navegador (“**http://localhost:8090**”). O usuário padrão é “**admin**” com senha “**admin**”. Repare que, ao salvar o seu arquivo pelo Windows, terá efeito imediato na máquina virtual. Na Figura 2.3 é mostrada a tela de *login* da sua nova instância.

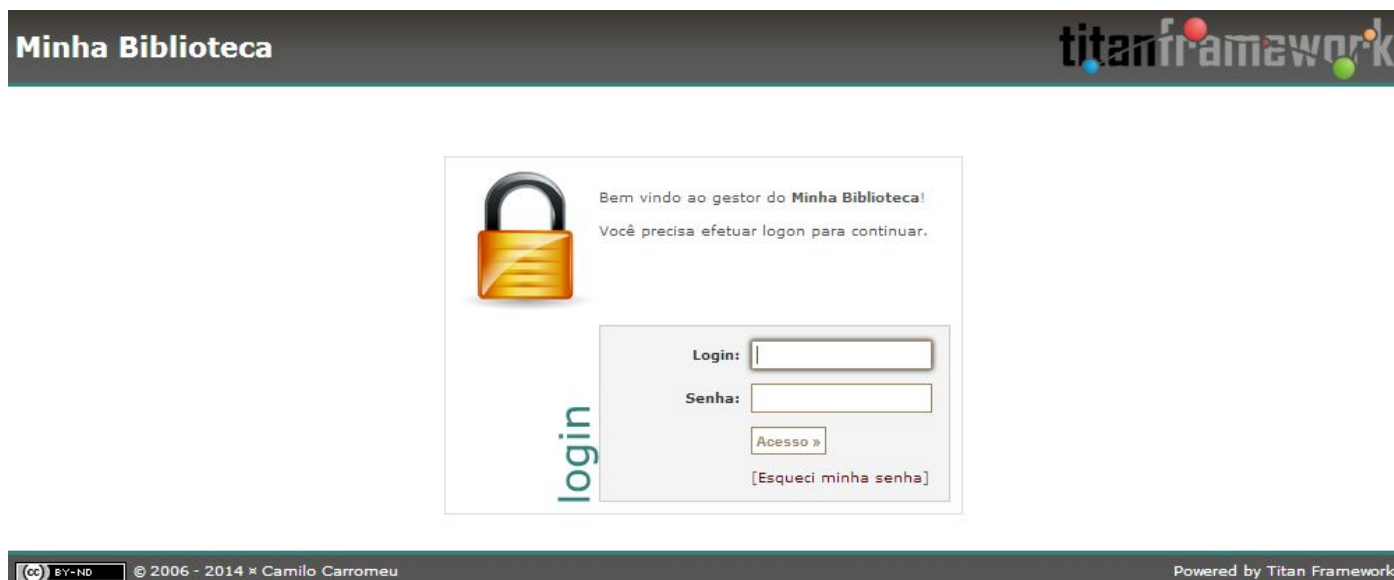


Figura 2.3: Tela de *login* da nova instância.

¹⁸ <https://groups.google.com/d/topic/titan-framework/ZDopSu-FKFQ/discussion>

Agora a sua instância está configurada e funcional, e você pode iniciar o desenvolvimento do sistema final incluindo a ela seus próprios requisitos. Para o nosso exemplo “**library**”, vamos começar criando um CRUD¹⁹ para possibilitar a gestão de livros. No Titan, normalmente utilizamos o componente “**global.generic**” para a criação de CRUDs. Este componente espera poder gerenciar tabelas no banco de dados em um formato semelhante ao abaixo:

```
CREATE TABLE esquema.tabela (  
  
    // A chave primária normalmente será denominada 'id' e será um SERIAL  
    id SERIAL PRIMARY KEY,  
  
    // Os campos específicos serão criados para corresponder a cada tipo do Titan,  
    // conforme visto na primeira parte deste livro  
    campo1 TIPO(99),  
    campo2 TIPO(99),  
    ...  
    campoN TIPO(99),  
  
    // Campos mandatórios necessário ao registro de modificações (autor,  
    // data de criação da tupla e data da última modificação).  
    _user INTEGER NOT NULL,  
    CONSTRAINT tabela_user_fk FOREIGN KEY (_user) REFERENCES titan._user (_id)  
    ON UPDATE CASCADE ON DELETE RESTRICT,  
    _create TIMESTAMP WITH TIME ZONE NOT NULL DEFAULT now(),  
    _update TIMESTAMP WITH TIME ZONE NOT NULL DEFAULT now()  
);
```

Para, seguindo o modelo acima, criar suas próprias tabelas no banco de dados da instância, recomenda-se conectar nele utilizando um gerenciador para PostgreSQL. Um bom (e gratuito) gerenciador é o **EMS SQL Manager for PostgreSQL Freeware**²⁰. Após instalá-lo, conecte no PostgreSQL da máquina virtual (“**localhost**” na porta “**5431**” com login e senha “**titan**”) e mapeie o banco de dados da aplicação (denominado “**library**”). Você verá os artefatos que foram criados no momento em que instanciou a instância básica do Titan, conforme é mostrado na Figura 2.4.

¹⁹ <http://pt.wikipedia.org/wiki/CRUD>

²⁰ <http://www.sqlmanager.net/products/postgresql/manager/download>

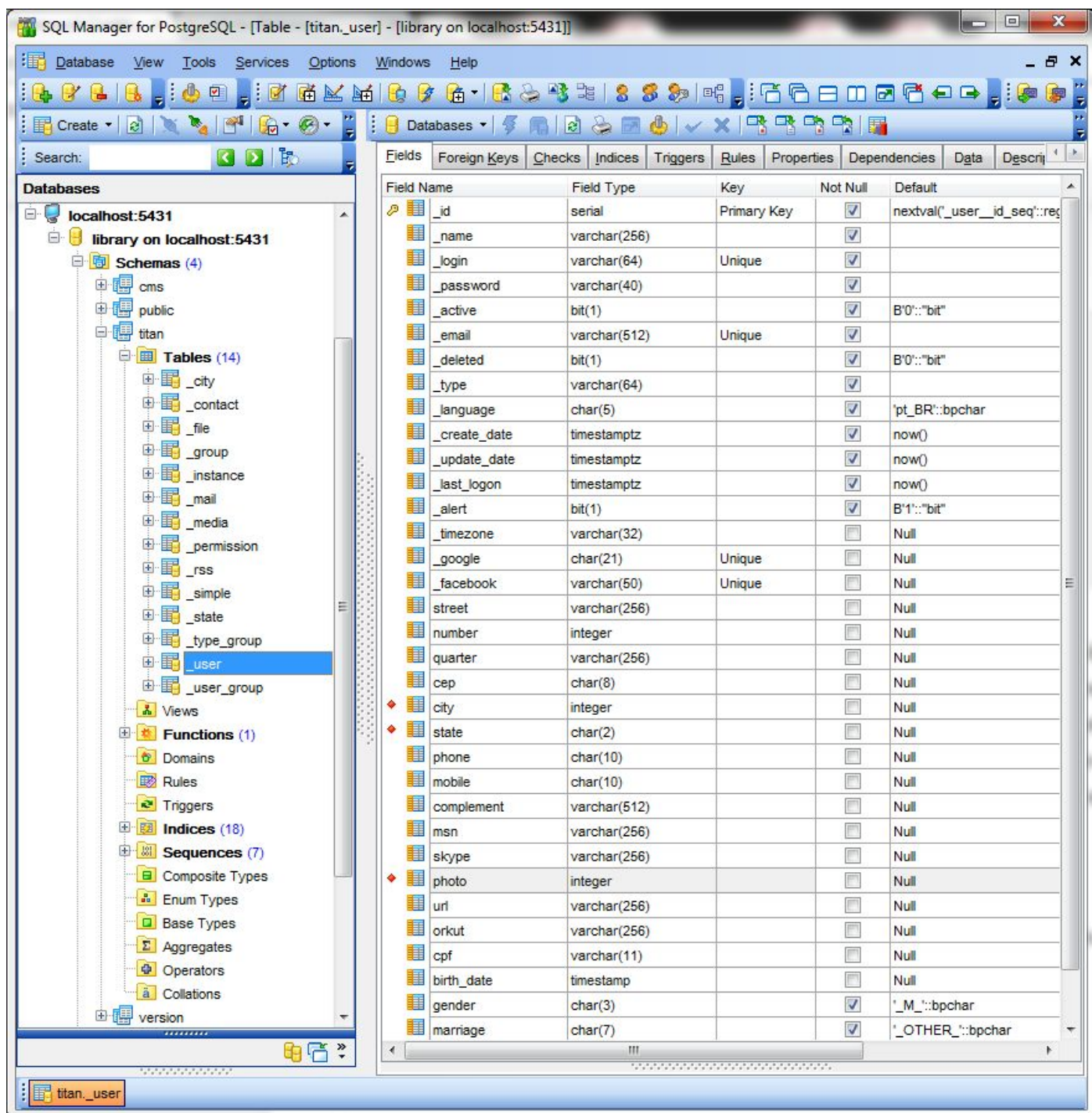


Figura 2.4: Tela do EMS com o banco de dados “*library*” após a instanciação.

Note que há apenas tabelas mandatórias do Titan por enquanto (àquelas prefixadas pelo caracter *underscore*). Todas elas ficam no *schema* denominado “**titan**”, que é o *schema* principal da aplicação (conforme estipulado no atributo “schema” da tag “database” do arquivo

“configure/titan.xml”). Além deste *schema*, temos três outros:

- “**cms**” é criado por padrão como sugestão de onde ficarão as tabelas específicas da aplicação (não-mandatórias). Podemos renomear este *schema* para um nome mais relacionado ao domínio da aplicação, tal como “**library**” para o nosso exemplo;
- “**public**” é criado automaticamente pelo PostgreSQL em todo novo banco de dados. Pode ser apagado pois não utilizaremos; e
- “**version**” é o *schema* onde o Titan irá gerar os artefatos necessários ao controle de versões de conteúdo de seções (conforme estipulado no atributo “*schema*” da *tag* “version-control” do arquivo “configure/titan.xml”). O uso desta funcionalidade será explicada no livro “**Cookbook Master Chef**”.

Podemos agora fazer as alterações necessárias no banco de dados para que ele comporte as seções da nossa aplicação de biblioteca. Primeiramente, conforme adiantado, iremos alterar o nome do *schema* “**cms**” para “**library**”:

```
ALTER SCHEMA cms RENAME TO library;
```

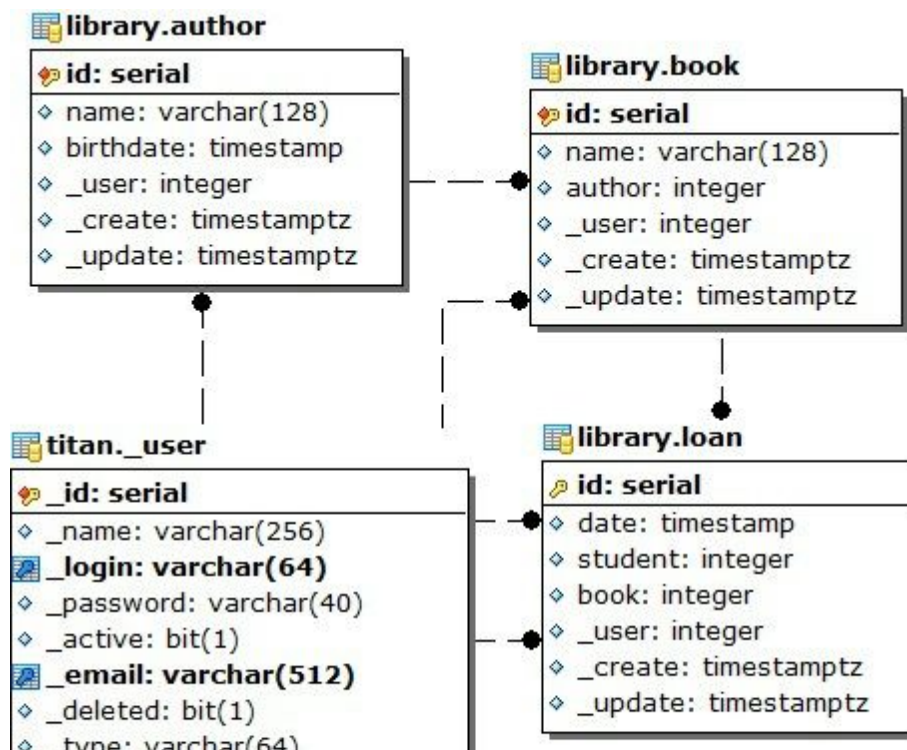


Figura 2.5: DER com as tabelas específicas criadas para o sistema de bibliotecas.

As novas tabelas a serem inseridas no banco de dados, mostradas no diagrama entidade-relacionamento da Figura 2.5, podem ser criadas por meio dos seguintes comandos:

```
CREATE TABLE library.author (
  id SERIAL PRIMARY KEY,
  name VARCHAR(128) NOT NULL,
  birthdate TIMESTAMP NOT NULL,
  _user INTEGER NOT NULL,
  CONSTRAINT author_user_fk FOREIGN KEY (_user) REFERENCES titan._user (_id)
  ON UPDATE CASCADE ON DELETE RESTRICT,
  _create TIMESTAMP WITH TIME ZONE NOT NULL DEFAULT now(),
  _update TIMESTAMP WITH TIME ZONE NOT NULL DEFAULT now()
);

CREATE TABLE library.book (
  id SERIAL PRIMARY KEY,
  name VARCHAR(128) NOT NULL,
  author INTEGER NOT NULL,
  CONSTRAINT book_author_fk FOREIGN KEY (author) REFERENCES library.author (id)
  ON UPDATE CASCADE ON DELETE RESTRICT,
  _user INTEGER NOT NULL,
  CONSTRAINT book_user_fk FOREIGN KEY (_user) REFERENCES titan._user (_id)
  ON UPDATE CASCADE ON DELETE RESTRICT,
  _create TIMESTAMP WITH TIME ZONE NOT NULL DEFAULT now(),
  _update TIMESTAMP WITH TIME ZONE NOT NULL DEFAULT now()
);

CREATE TABLE library.loan (
  id SERIAL PRIMARY KEY,
  date TIMESTAMP NOT NULL,
  student INTEGER NOT NULL,
  CONSTRAINT loan_student_fk FOREIGN KEY (student) REFERENCES titan._user (_id)
  ON UPDATE CASCADE ON DELETE RESTRICT,
  book INTEGER NOT NULL,
  CONSTRAINT loan_book_fk FOREIGN KEY (book) REFERENCES library.book (id)
  ON UPDATE CASCADE ON DELETE RESTRICT,
  _user INTEGER NOT NULL,
  CONSTRAINT loan_user_fk FOREIGN KEY (_user) REFERENCES titan._user (_id)
  ON UPDATE CASCADE ON DELETE RESTRICT,
  _create TIMESTAMP WITH TIME ZONE NOT NULL DEFAULT now(),
  _update TIMESTAMP WITH TIME ZONE NOT NULL DEFAULT now()
);
```

Observe que toda tabela criada possui as colunas:

- “**_user**”: Uma chave estrangeira para a tabela mandatória de usuários da aplicação (no Titan, todos os usuários sempre estarão na tabela “**titan._user**”). Esta coluna irá referenciar o último usuário que criou ou alterou a tupla;
- “**_create**”: A data de criação da tupla. Repare que esta coluna é do tipo *timestamp* com

fuso horário (*time zone*), o que é essencial para que o horário de criação seja apresentado de forma adequada ao usuário; e

- “**_update**”: A data da última alteração na tupla.

Note o uso das convenções de código recomendadas para banco de dados, já citadas na seção de “Boas Práticas de Programação” da primeira parte do livro.

Neste primeiro exemplo, todas as seções da nossa aplicação serão CRUDs, ou seja, todas serão renderizadas pelo componente “**global.generic**”. Começaremos pela gestão de autores (tabela “**author**”). Inicialmente, precisamos criar uma pasta dentro do diretório “section” da raiz onde irão ficar os arquivos de marcação XML que parametrizam a seção. Para nosso propósito, o nome “**author**” para a pasta é apropriado.

Uma vez que a pasta da seção esteja criada, é necessário declarar ao Titan que há uma nova seção. Conforme já mencionado, as seções do titan são declaradas no arquivo “**configure/business.xml**” que, na instância básica, tem o seguinte conteúdo:

```
<?xml version="1.0" encoding="UTF-8"?>
<section-mapping>
  <section
    label="Home | pt_BR: Página Inicial"
    name="home"
    component="global.home"
    default="true"
  />

  <section
    label="Base of Files | pt_BR: Banco de Arquivos"
    name="archive"
    component="global.archive"
  />

  <section
    label="Access Control | pt_BR: Controle de Acesso"
    name="access"
    component="global.group"
    admin="true"
  />

  <section
    label="Managers Users | pt_BR: Usuários Gestores"
    name="manager"
    component="global.userPrivate"
    father="access"
  />
</section-mapping>
```

Repare que já estão ativas as seguintes seções na instância básica: “**home**”, que é a seção padrão, ou seja, a que é carregada por padrão ao iniciar a aplicação, e onde o usuário pode visualizar e editar seu perfil (*profile*); “**archive**”, onde é possível visualizar e gerir todos os arquivos enviados por usuários à aplicação (por *upload*); “**access**”, onde é possível gerenciar os grupos de usuários e permissões, conforme explicitado na seção “Usuários e Permissões” da primeira parte deste livro; e, “**manager**”, que permite gerenciar o único tipo de usuário disponível na instância básica.

Para mapear uma nova seção neste arquivo basta, portanto, adicionar uma nova *tag* “**section**”. Esta *tag* aceita os seguintes atributos:

- **label:** Será o rótulo, ou seja, o nome descritivo que aparecerá para os usuários do sistema. Repare que o valor deste atributo aceita internacionalização (*i18n*).
- **name:** O nome Unix da seção. Deve ser homônimo à pasta criada no diretório “**section**” (da raiz da instância) que abrigará os XMLs que parametrizam as *engines*.
- **component:** O componente que irá renderizar a seção. Caso seja um componente nativo, o valor deste atributo pode ser simplesmente o nome do componente (por exemplo, “**global.generic**” ou “**global.userProtected**”). Caso seja um componente local, o valor deverá ser o caminho até a pasta do componente (por exemplo, “**local/component/library.loan/**”). Caso não seja declarado este atributo, o Titan considerará que trata-se de uma “**seção falsa**” (*fake*), utilizada apenas para agrupar outras seções no menu.
- **father:** O nome Unix de uma “seção-pai”. Objetiva organizar o menu da aplicação em níveis hierárquicos, mas não têm impacto nas funcionalidades.
- **description:** Descrição. Irá aparecer apenas no manual do usuário gerado. O valor deste atributo aceita internacionalização (*i18n*).
- **doc:** Informações adicionais que irão aparecer apenas no manual do usuário gerado. O valor deste atributo aceita internacionalização (*i18n*).
- **admin:** Caso este atributo seja declarado na *tag* da seção com o valor ‘*true*’, o Titan irá entender que esta seção e todas as suas ações deverão estar acessíveis aos usuários de grupos administradores (independentemente das permissões de fato concedidas). Repare

que no código acima a seção “**access**” possui o atributo declarado. Com isso, se você logar na aplicação com o usuário “**admin**”, irá perceber que todas as seções estão inacessíveis, com exceção do “**Controle de Acesso**”.

- **hidden:** Caso este atributo seja declarado na *tag* da seção com o valor 'true', o Titan irá entender que esta seção não deverá aparecer no menu. É útil quando se necessita criar seções apenas para habilitar *scripts* ou *scheduler jobs*²¹ do componente (funcionalidades que serão detalhadas no “**Cookbook Master Chef**”).

Pronto, agora você está apto a declarar sua nova seção no “**business.xml**”. Vamos também criar uma seção falsa (*fake*) denominada “Biblioteca”, de forma a agrupar no menu nossas seções específicas. Recomendamos colocá-las após a declaração da “home”:

```
<section
  label="Library | pt_BR: Biblioteca"
  name="library"
/>

<section
  label="Authors | pt_BR: Autores"
  name="author"
  component="global.generic"
  father="library"
/>
```

Após mapear a nova seção, crie um arquivo chamado “**config.inc.xml**” em seu diretório (“**section/author/**”). Esse arquivo vai conter as configurações das ações dentro da seção. Seu conteúdo será o seguinte:

```
<?xml version="1.0" encoding="UTF-8"?>
<action-mapping>

  <action
    name="list"
    label="List Authors | pt_BR: Listar Autores"
    default="true"
    description="">
    <menu action="create" />
    <menu function="search" />
  </action>

  <action
    name="view"
    label="View Author | pt_BR: Visualizar Autor"
```

²¹ <https://groups.google.com/d/topic/titan-framework/DDQKpGUgDPc/discussion>

```
        description="">
        <menu action="list" />
        <menu action="edit" />
        <menu function="print" />
    </action>

    <action
        name="create"
        label="Create New Author | pt_BR: Criar Novo Autor"
        description="">
        <menu function="save" />
        <menu action="list" image="close.png" />
    </action>

    <action
        name="edit"
        label="Edit Author | pt_BR: Editar Autor"
        description="">
        <menu function="save" />
        <menu action="list" image="close.png" />
    </action>

    <action
        name="delete"
        label="Delete Author | pt_BR: Apagar Autor"
        description="">
        <menu function="delete" />
        <menu action="list" image="close.png" />
    </action>
</action-mapping>
```

Observe o código. Note que as possíveis ações dos usuários estão mapeadas por meio da tag “**action**” dentro da tag “**action-mapping**”. Essa tag possui os atributos:

- **name:** O nome da ação. Cada ação deve ter um nome único. No momento de carregar o modelo de dados, o Titan irá procurar por um arquivo XML na pasta da seção homônimo à ação.
- **engine:** O nome do motor que irá renderizar a ação. Quando não especificado, o Titan irá considerar que a *engine* é homônima à ação.
- **label:** O rótulo (ou nome descritivo da ação) que será visualizado pelo usuário do sistema. Suporta internacionalização (*i18n*).
- **default:** Aceita os valores 'true' ou 'false', determinando se esta é a ação padrão, ou seja, àquela que será chamada quando o usuário acessa a seção a partir do menu da aplicação.

- **description:** Descrição da ação. Será especialmente útil na geração do manual do usuário.
- **path:** É possível sobrescrever a engine do componente passando neste atributo o caminho para uma *engine* em uma pasta local da aplicação. Por convenção, engines customizadas devem ser colocadas no diretório denominado “**local**” da raiz da instância, em um subdiretório com o mesmo nome da seção dentro de outro subdiretório denominado “**custom**”. Por exemplo, para sobrescrever a engine “**list**” da seção “**author**” do nosso sistema de bibliotecas, poderíamos criar os *scripts* de *prepare* e *view* no diretório “**local/custom/author/**” (o atributo “**path**” receberia, portanto, este caminho como valor).
- **warning:** Recebe um aviso para ser exibido no topo da seção. Também gera conteúdo no manual do usuário.
- **xml-path:** Caso queira forçar para que a *engine* receba outro modelo de dados que não aquele do XML homônimo à ação, pode-se forçar por meio deste atributo o XML que ela deverá considerar.
- **menu:** É uma lista de *tags* internas que especifica o menu da ação, ou seja, aquela coleção de botões que aparece no canto superior esquerdo, abaixo do menu do sistema, conforme mostrado na Figura 2.6.

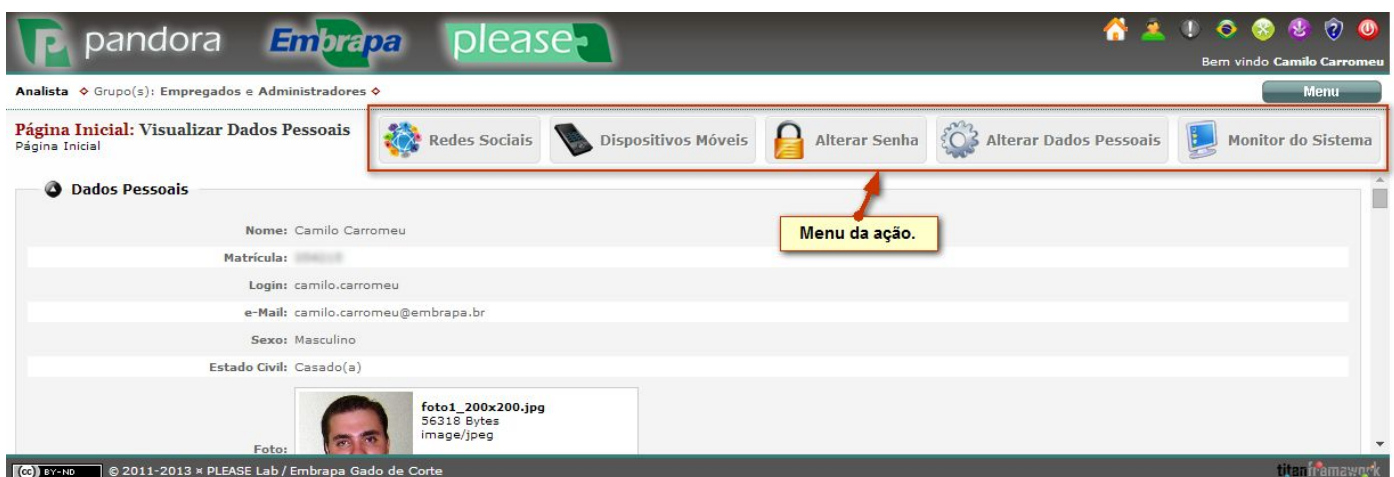


Figura 2.6: Menu da ação.

A tag “**menu**” também mapeia artefatos do repositório de ativos do Titan (as opções

possíveis do menu são renderizadas por artefatos, assim como os componentes e os tipos). Desta forma, os atributos desta *tag* parametrizam estes artefatos, possibilitando adicionar funcionalidades à ação. Todas as ações de menu nativas serão explicadas no “**Cookbook Master Chef**”, mas no âmbito deste exemplo as mais interessantes são:

- **Abrir outra ação:** Pode-se colocar um botão para abrir outra ação da mesma ou de outra seção. O formato (com os atributos correspondentes) deve ser:

```
<menu section="author" action="list" label="Authors | pt_BR: Autores" image="list.png" />
```

O atributo “**section**”, quando não especificado, será a seção atual. O “**label**”, quando não especificado, será o rótulo da ação apontada. O atributo “**image**” é o ícone que aparece do lado esquerdo do botão e as opções nativas são mostradas na Figura 2.7.

- **Imprimir ação:** Pode-se colocar um botão que permite imprimir a ação atual:

```
<menu function="print" label="Print | pt_BR: Imprimir" image="print.png" />
```

- **Salvar formulário:** Pode-se colocar um botão que permite salvar o formulário da ação atual (para *engines* que tenham *commit*, tal como “*create*” ou “*edit*”):

```
<menu function="save" label="Save | pt_BR: Salvar" image="save.png" />
```

- **Apagar item:** Pode-se colocar um botão que permite apagar o item carregado (para *engines* que tenham suporte, tal como “*delete*”):

```
<menu function="delete" label="Delete | pt_BR: Apagar" image="delete.png" />
```

- **Exportar CSV:** Pode-se colocar um botão que permite extrair uma ação de listagem em formato CSV (para *engines* que tenham suporte, tal como “*list*”):

```
<menu function="csv" label="Export CSV | pt_BR: Exportar CSV" image="csv.png" />
```

- **Buscar itens:** Pode-se colocar um botão que permite filtrar os itens listados com base em critérios de busca (para *engines* que tenham suporte, tal como “list”):

```
<menu function="search" label="Search | pt_BR: Buscar" image="search.png" />
```

- **Executar JavaScript:** Pode-se colocar um botão que permite executar funções JavaScript, desde que disponibilizadas pelo desenvolvedor. Por exemplo, alguns componentes nativos tem suporte à exportação CSV em dois passos, que permite selecionar os campos a serem exportados:

```
<menu function="js" js="exportCsv ()" image="csv.png" label="Fazer download do arquivo CSV" />
```



Figura 2.7: Ícones nativos das ações de menu.

Depois do arquivo “**config.inc.xml**” criado (com as ações da seção devidamente mapeadas), precisamos criar os arquivos que definem o modelo de dados de cada ação. Conforme vimos, a *engine* da ação procura por um arquivo XML homônimo à ela na pasta da seção. Se observarmos novamente o conteúdo do “**config.inc.xml**”, nossa primeira ação (a ação padrão,

pois tem o atributo “**default**” com valor 'true') é denominada “**list**”, ou seja, quando o usuário acessa a seção o Titan irá chamar esta ação procurando por um arquivo denominado “**list.xml**” na pasta da seção com o modelo de dados a ser carregado. Vamos criar então este arquivo com o seguinte conteúdo:

```
<?xml version="1.0" encoding="UTF-8"?>
<view table="library.author" primary="id" paginate="15">
  <field type="Phrase" column="name" label="Nome" />
  <field type="Date" column="birthdate" label="Data de Nascimento" />
  <icon action="view" image="view.gif" label="Visualizar Autor" default="true" />
  <icon action="edit" image="edit.gif" label="Editar Autor" />
  <icon action="delete" image="delete.gif" label="Apagar Autor" />
</view>
<search table="library.author">
  <field type="Phrase" column="name" label="Nome" />
  <field type="Date" column="birthday" label="Data de Nascimento" />
</search>
```

Observe o código acima. Ações de *engines* “**list**” normalmente suportam duas *tags* no nível da raiz: “**view**” e “**search**”. A primeira será o modelo de dados da listagem de itens onde, no exemplo acima, podemos verificar que há dois campos (*tag* “**field**”) declarados: o que mapeia a coluna “**name**” (do tipo “Phrase”) e o que mapeia a coluna “**birthdate**” (do tipo “Date”). Além disso, após a declaração dos campos, há a declaração de ícones de ação (*tag* “**icon**”).

Analista ♦ Grupo(s): Empregados e Administradores ♦

Notícias: Listar Itens
Portal da Unidade » Notícias

Menu

+ Criar Itens 🔍 Buscar Itens 📡 RSS Feed

Título	Publicado?	Data	Última Modificação	Autor	
Descarte de vacas é essencial para manter produtividade nas propriedades	Sim	27-05-2014	27-05-2014 13:13	Kadijah Suleiman	Ícones de ação.
Entrega de obras nos 37 anos de Gado de Corte	Sim	21-05-2014	22-05-2014 10:36:25	Dalizia Montenario de Aguiar	
Produtor deve registrar sua propriedade no CAR	Sim	14-05-2014	14-05-2014 15:28:14	Dalizia Montenario de Aguiar	
Técnicos e produtores da Cocamar recebem informações sobre sistemas integrados	Sim	09-05-2014	09-05-2014 15:54:56	Kadijah Suleiman Jaghub	

(cc) BY-ND © 2011-2013 x PLEASE Lab / Embrapa Gado de Corte titanframework

Figura 2.8: Ícones de ação na listagem de itens.

Os ícones de ação possibilitam ao usuário acessar ações e funcionalidades relacionadas diretamente ao item da listagem (por exemplo, editá-lo ou apagá-lo). Na Figura 2.8 é mostrada a disposição destes ícones na listagem de itens. Assim como os componentes, tipos e menus de ação, os ícones de ação são conjuntos de artefatos de código que fazem parte do repositório de ativos do Titan. Ou seja, assim como todos os demais ativos, eles podem ser estendidos por programadores para serem customizados.

Os ícones de ação nativos permitem:

- **Chamar uma ação:** Pode-se colocar um ícone que efetua a chamada para uma ação específica:

```
<icon
  id="_VIEW_"
  action="view"
  section="author"
  image="view.gif"
  label="View Author | pt_BR: Visualizar Autor"
  image="view.gif"
  default="true"
/>
```

O atributo “**id**” é um identificador único para o ícone e será necessário declará-lo apenas quando se quer pegar o objeto instanciado no código do componente. O atributo “**action**” é o nome da ação para onde o usuário será enviado. O atributo “**section**” é a seção que contém a ação, mas só é necessário declará-lo quando a ação não é da seção atual. O atributo “**image**” serve para alterar imagem do ícone (as imagens nativas disponíveis são mostradas na Figura 2.9). Caso não seja declarada, o Titan tenta carregar uma imagem de ícone homônima à ação. Por fim, o atributo “**default**”, quando declarado com o valor ‘true’, determina que a ação deste ícone será a ação padrão de toda a linha da listagem, ou seja, os campos da listagem serão *links* para a ação deste ícone.



Figura 2.9: Imagens nativas dos ícones de ação.

- **Chamar uma ação *in place*:** Pode-se colocar um ícone que efetua a chamada para uma ação específica porém, conforme mostrado na Figura 2.10, ao invés de recarregar a página, a ação é aberta em um *iframe* logo abaixo da linha:

```
<icon
  function="[ajax]"
  action="edit"
  image="edit.gif"
  label="Edit | pt_BR: Editar"
  default="true"
/>
```

Aplicativo Móvel de Manejo Pecuário
Bem vindo Camilo Carromeu

Usuário
Grupo(s): Pecuaristas e Administradores
Menu

Pelagens: Listar Itens

Configurações » Pelagens

+ Criar Itens

Descrição	Código	Cor	Ativo	Última Alteração	Último Autor
Amarela	AM		Sim	06-12-2013 13:06:17	Camilo Carromeu

*Descrição: 121

*Cor: 85910C

Salvar

Araçá	AR		Sim	06-12-2013 10:56:05	Camilo Carromeu
Baia	BA		Sim	06-12-2013 10:56:05	Camilo Carromeu
Branca	BR		Sim	06-12-2013 10:56:05	Camilo Carromeu

© 2006 - 2014 * Camilo Carromeu

Figura 2.10: Abrindo uma ação de edição *in place*.

- **Alterar o *status* do item:** Pode-se, por meio de um ícone de ação, alterar o valor de uma coluna booleana do item:

```
<icon
  function="[status]"
  image="confirm.gif"
  label="Ativar/Desativar"
  id="_ACTIVE_"
  table="taurus.blood_degree"
  primary="id"
  column="active"
  message="&lt;span style=&quot;color: #900; font-weight: bold;&quot;&gt;Atenção!&lt;/span&gt; Tem
certeza que deseja &lt;b&gt;ALTERAR O ESTADO&lt;/b&gt; deste item?">
  <status value="1" label="Ativar" color="#090" />
  <status value="0" label="Desativar" color="#900" />
</icon>
```

Repare que será possível controlar aspectos da usabilidade desta funcionalidade, tal como a cor em função do estado, conforme é mostrado na Figura 2.11.



Figura 2.11: Ícone de ação para alterar o *status* do item.

- **Chamar uma função JavaScript:** Você pode declarar no atributo “**function**” o nome de uma função JavaScript implementada no componente para que seja executada pelo usuário (no momento da chamada, será passada à função o valor da chave primária do item e o objeto JavaScript da linha da listagem):

```
<icon function="showGallery" label="Visualizar Fotos e Vídeos" image="camera.gif" />
```

- **Duplicar o item:** Pode-se duplicar o item por meio deste ícone de ação (neste caso, o item é duplicado e o fluxo da aplicação leva o usuário à tela de edição, para que ele efetue as alterações necessárias no novo item):

```
<icon function="[copy]" action="edit" label="Duplicar" />
```

Podemos agora criar os modelos de dados das demais ações de nossa seção, ou seja, as ações “**view**”, “**create**”, “**edit**” e “**delete**” declaradas em nosso “**config.inc.xml**”. Conforme explicitado, devemos criar um XML homônimo para cada uma destas ações. Entretanto, como muitas vezes estes arquivos serão idênticos, o Titan possibilita que seja criado um arquivo denominado “**all.xml**”. Quando o arquivo homônimo à ação não é encontrado, o Titan procura por este. É muito inteligente fazer uso desta característica, ou seja, criar apenas um “**all.xml**” e criar arquivos específicos para as ações apenas quando houverem especificidades. Esta técnica reduz enormemente o ônus de manutenções corretivas e evolutivas.

Vamos então criar o “**all.xml**”, na pasta da seção “**author**”, com o seguinte conteúdo:

```
<?xml version="1.0" encoding="UTF-8"?>
<form table="library.author" primary="id">
  <go-to flag="success" action="[default]" />
  <go-to flag="fail" action="[same]" />
  <group label="Dados do Autor">
    <field type="Phrase" column="name" label="Name | pt_BR: Nome" max-length="128" />
    <field type="Date" column="birthdate" label="Birthdate | pt_BR: Data de Nascimento"
    first-year="1900" last-year="[now]" />
  </group>
</form>
```

Observe o código. Ao contrário da *engine* “**list**”, as *engines* “**view**”, “**create**”, “**edit**” e “**delete**” esperam receber como modelo de dados um arquivo XML cuja tag-raiz é denominada “**form**”. Esta *tag* contém os atributos “**table**” (nome da tabela no banco de dados) e “**primary**” (nome da chave primária da tabela).

Internamente, teremos primeiro a *tag* “**go-to**”. Esta *tag* é responsável pelo controle do

fluxo da ação. Contém os atributos: “**flag**” e “**action**”, que definem o que acontece com a página caso a operação seja bem sucedida ou falhe. Repare que o atributo “**action**” pode receber o nome de uma ação ou as palavras reservadas “[**default**]” (ação padrão da seção) ou “[**same**]” (a ação atual). No código acima, em caso de falha no salvamento, o fluxo leva o usuário ao mesmo formulário, apresentando uma mensagem de erro. Em caso de sucesso, é apresentado ao usuário a ação “*list*”, ou seja, a que têm o atributo “**default**” declarado como ‘*true*’.

Em seguida, teremos a tag “**group**”, que permite agrupar os campos do modelo de dados. Esta tag possui os tributos:

- **label:** O rótulo (ou nome descritivo do grupo) que será visualizado pelo usuário do sistema. Suporta internacionalização (*i18n*).
- **display:** Aceita os valores ‘*visible*’ e ‘*hidden*’ para determinar se o grupo deve ser apresentado de forma expandida ou fechada, respectivamente.
- **info:** Uma informação que aparece em destaque no topo do grupo (antes dos campos), conforme mostrado na Figura 2.12.



Figura 2.12: Uso do atributo “**info**” da tag “**group**”.

No interior da tag “**group**” são colocados os campos (*fields*) mapeando as colunas da tabela. Cada tipo de “**field**” (e os atributos aceitos) são explicitados na primeira parte deste livro

(na seção sobre os “Tipos” do Titan). Repare que os campos não precisam estar, necessariamente, dentro de grupos e que podemos ter, portanto, nenhum ou diversos grupos.

Pronto. Caso você acesse o sistema agora (utilizando o login e senha “**admin**”) verá que a nova seção já aparece no menu, mas está desabilitada. Isto ocorre pois o usuário não têm permissão de acesso à ela. Navegue no menu até “**Controle de Acesso » Controle de Acesso**”. Iremos conceder permissões para o único grupo existente, denominado “**Administradores**”, do qual seu usuário faz parte. Para isto, na tela que aparece, clique sobre o ícone de ação em forma de cadeado na linha do grupo, conforme mostrado na Figura 2.13.

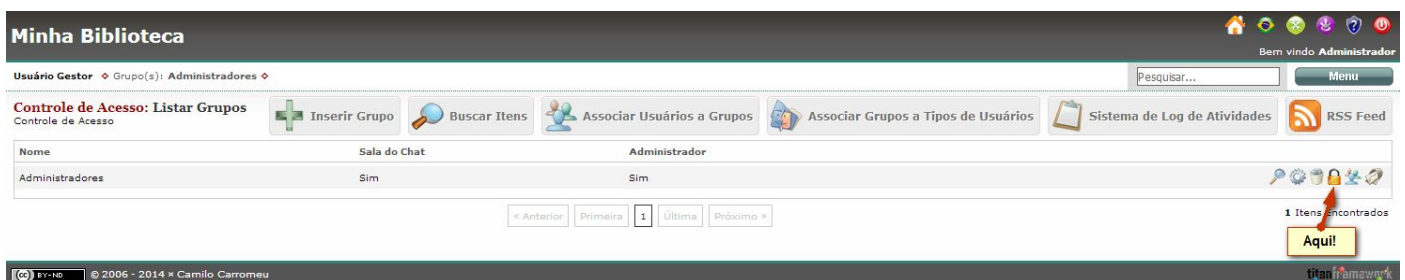


Figura 2.13: Acessando as permissões de um grupo de usuários.

Em seguida, selecione todas as ações listadas de todas as seções e salve, concedendo, desta forma, todas as permissões ao grupo “**Administradores**”, conforme mostrado na Figura 2.14.

Minha Biblioteca

Bem vindo **Administrador**

Usuário Gestor ♦ Grupo(s): **Administradores** ♦

Controle de Acesso: Associar Permissões a Grupos
 Controle de Acesso

Permissões vinculadas com sucesso.

Permissões de Acesso às Seções do Sistema

☒ Permissão de acesso à seção **Página Inicial**

☒ Permissão de acesso à seção **Biblioteca » Autores**

Acesso às Ações da Seção Autores

☒ Selecionar Todas

☒ Listar Autores

☒ Visualizar Autor

☒ Criar Novo Autor

☒ Editar Autor

☒ Apagar Autor

Permissões da Seção Autores

☐ Selecionar Todas

☒ Permissão de acesso à seção **Banco de Arquivos**

☒ Permissão de acesso à seção **Controle de Acesso**

☒ Permissão de acesso à seção **Controle de Acesso » Usuários Gestores**

© 2006 - 2014 ♦ Camilo Carromeu

Figura 2.14: Concedendo permissões de acesso ao grupo de usuários.

Você irá perceber que todas a seções do menu são agora clicáveis. Navegue pelo menu até **“Biblioteca » Autores”** e comece a cadastrar autores no seu sistema (efetue todos os testes que desejar, listando e buscando autores, criando novos e editando e apagando os existentes).

Repita agora os passos para criar as seções **“Livros”** e **“Empréstimos”**. Dica: você precisará utilizar campos do tipo **“Select”** para relacionar um autor a um livro e um estudante (usuário do sistema) e um livro a um empréstimo.

2.3 Criação de Tipos e Componentes

Conforme vimos, o Titan oferece uma coleção de tipos e componentes nativos para desenvolvimento de aplicações diversas, mas também facilita a criação e derivação destes e de outros artefatos de seu repositório. Muitas vezes será preciso fazer alterações em alguns componentes, tipos, ícones de ação e/ou botões de menu de ação nativo para atender alguma funcionalidade específica de nossas aplicações.

Voltando ao exemplo do sistema de biblioteca, digamos que queremos listar todas as operações de empréstimos realizadas pelo usuário logado no momento. Repare que, utilizando o “**global.generic**”, podemos conceder ou restringir o acesso do usuário a uma determinada ação, mas não a parte dos dados listados. Assim, fica claro que precisaremos modificar a *engine* de listagem, ou seja, será necessário criar um novo componente derivado do “**global.generic**” que tenha o suporte que precisamos.

2.3.1 Criando um Componente Derivado

Queremos que, ao acessar a ação “**list**” da seção “**loan**” (“**Empréstimo**”), sejam listados apenas as operações de empréstimos que foram realizadas pelo usuário logado naquele momento, e não todas as operações de empréstimos realizadas no sistema. Iniciaremos o processo criando uma permissão no controle de acesso referente à esta restrição. Isso pode ser feito acrescentando a tag “**permission**” no arquivo “**config.inc.xml**” da seção “**loan**”:

```
<?xml version="1.0" encoding="UTF-8"?>
<action-mapping>

    <permission name="_VIEW_ALL_" label="Pode visualizar todos os empréstimos." />

    <action
        name="list"
        label="Listar Empréstimo"
        default="true">
        <menu action="create" />
        <menu function="search" />
    </action>

    <action
        name="view"
        label="Visualizar Empréstimo">
        <menu action="list" />
        <menu action="edit" />
        <menu function="print" />
    </action>

    <action
```

```
        name="create"
        label="Criar Novo Empréstimo">
        <menu function="save" />
        <menu action="list" image="close.png" />
    </action>

    <action
        name="edit"
        label="Editar Empréstimo">
        <menu function="save" />
        <menu action="list" image="close.png" />
    </action>

    <action
        name="delete"
        label="Apagar Empréstimo">
        <menu function="delete" />
        <menu action="list" image="close.png" />
    </action>

</action-mapping>
```

Note que a *tag* “**list**” tem os atributos “**name**” (um identificador único, dentro da seção, para a permissão) e “**label**” (uma descrição que será apresentada no controle de acesso). Conforme é mostrado na Figura 2.15, apenas o fato de colocar esta *tag* no “**config.inc.xml**” já faz com que se possa atribuir esta permissão a um grupo de usuários.

Minha Biblioteca

Bem vindo Administrador

Usuário Gestor

Grupo(s): Administradores

Pesquisar...

Menu

Controle de Acesso: Associar Permissões a Grupos

Controle de Acesso

Salvar

Listar Grupos

Permissões de Acesso às Seções do Sistema

☒ Permissão de acesso à seção **Página Inicial**

☒ Permissão de acesso à seção **Biblioteca » Autores**

☒ Permissão de acesso à seção **Biblioteca » Livros**

☒ Permissão de acesso à seção **Biblioteca » Estudantes**

☒ Permissão de acesso à seção **Biblioteca » Empréstimos**

Acesso às Ações da Seção Empréstimos

☐ Selecionar Todas

☒ Listar Empréstimos

☒ Visualizar Empréstimo

☒ Criar Novo Empréstimo

☒ Editar Empréstimo

☒ Apagar Empréstimo

Permissões da Seção Empréstimos

☐ Selecionar Todas

☒ Pode visualizar todos os empréstimos.

Permissão específica da seção.

☒ Permissão de acesso à seção **Banco de Arquivos**

☒ Permissão de acesso à seção **Controle de Acesso**

☒ Permissão de acesso à seção **Controle de Acesso » Usuários Gestores**

CC BY-ND

© 2006 - 2014 x Camilo Carromeu

titanframework

Figura 2.15: Criação de uma permissão específica para uma seção.

Desta forma, queremos que somente usuários pertencentes a grupos que tenham esta permissão possam visualizar todas as operações de empréstimo. Caso contrario, listará apenas os empréstimos que ele realizou. Entretanto, para que a regra de negócio tenha efeito, não basta apenas adicionar a permissão. É preciso fazer alterações no componente para que a permissão seja considerada. Ou seja, precisaremos criar um componente próprio derivado do componente inicialmente utilizado, o “**global.generic**”.

No diretório raiz da instância, crie uma nova pasta denominada “**local**” e, dentro dela, crie

©2014-2016 Titan Framework
http://titan.carromeu.com

Página 71 de 87
Versão 0.7 de 11/1/16

outra pasta chamada “**component**”. Neste diretório, por convenção, ficarão todos os componentes locais feitos especificamente para esta aplicação.

Lembrando a arquitetura do Titan, já comentada na primeira parte, o componente é um diretório com *scripts* PHP. Dentro dele encontraremos os motores (*engines*) que renderizam as ações. Cada *engine* é formada por um conjunto de até 3 (três) arquivos: o *view* (que não possui sufixo), o *prepare* e o *commit*. No caso de ações de listagem, suas *engines* raramente terão o *script* de *commit*, uma vez que não há submissão de formulários (isto também ocorre em ações de visualização de itens). Assim, se abrirmos, dentro do núcleo do Titan, a pasta do componente “**global.generic**” (“core/repos/component/global.generic/”), veremos que existem os arquivos “**list.php**” e “**list.prepare.php**” compondo esta *engine*.

Os *scripts* de *prepare* (com o sufixo “**.prepare.php**”) são arquivos destinados ao carregamento e/ou validações do modelo de dados, onde pode-se desenvolver alguma lógica de negócio antes do objeto ser passado à camada de visualização. Já o “**list.php**” implementará a camada de visualização da ação, gerando código HTML, CSS e JavaScript.

Se observarmos agora, ainda na pasta do componente “**global.generic**”, a *engine* “**create**”, veremos que ela possui os arquivos: “**create.prepare.php**”, “**create.php**” e “**create.commit.php**”. Os dois primeiros funcionam como descrito para a *engine* “**list**”. Já o “**create.commit.php**” é responsável por realizar a higienização e o salvamento de formulários submetidos pela ação (normalmente pelo método “POST”). Após o processamento da submissão, este *script* também efetua a mudança do fluxo de controle, enviando o usuário para a ação correta.

Agora podemos iniciar a criação de nosso componente derivado. Para tal, faça o seguinte:

1. Crie uma pasta denominada “**library.loan**” dentro do diretório “**local/component/**” criado anteriormente na raiz da sua instância;
2. Exporte todos os arquivos e subdiretórios do “**global.generic**” para esta pasta:

```
C:\> svn export https://svn.cnpgc.embrapa.br/titan/core/repos/component/global.generic  
C:\projetos\library\local\component\library.loan
```

Atenção! Não copie e cole simplesmente o conteúdo da pasta diretamente do *core*. Se você fizer isso, estará copiando os arquivos “.svn” do Titan (pastas ocultas que controlam a *work copy*), ocasionando um grande transtorno no controle de versões da sua aplicação. Para copiar arquivos e pastas do Titan sempre utilize a funcionalidade “**Export**” do cliente Subversion (SVN).

Para utilizar o seu novo componente (ao invés do “**global.generic**”), basta editar o mapeamento da seção “**loan**” no arquivo “**configure/business.xml**”, trocando o valor do atributo “**component**” para o caminho até o diretório do novo componente:

```
<section
  label="Loans | pt_BR: Empréstimos"
  name="loan"
  component="local/component/library.loan/"
  father="library"
/>
```

Agora a seção “**loan**” (“Empréstimos”) usará o componente que acabamos de criar, mas seu comportamento será idêntico ao componente nativo “**global.generic**”, pois não foram feitas alterações nas *engines*. Para implementar a restrição na listagem, abra o arquivo “**list.prepare.php**” do seu componente (diretório “**local/component/library.loan/**”). O conteúdo deste arquivo deve ser semelhante ao seguinte:

```
<?
$search = new Search ('search.xml', 'list.xml');
$search->recovery ();

$view = new View ('list.xml');
if (!$view->load ($search->makeWhere ()))
    throw new Exception (__ ('Unable to load data!'));
?>
```

Modifique o arquivo da seguinte forma:

```
<?
$search = new Search ('search.xml', 'list.xml');
$search->recovery ();
```



```
$view = new View ('list.xml');

$where = "";

if (!$user->hasPermission ('_VIEW_ALL_'))
    $where = "_user = '". $user->getId () ."'";

if (!$view->load ($search->makeWhere ($where)))
    throw new Exception (__ ('Unable to load data!'));

?>
```

Salve o arquivo. Agora a ação “*list*” da seção “**loan**” listará apenas as operações de empréstimos realizadas pelo usuário logado no momento.

Existem outras formas mais elegantes de fazer este filtro, uma vez que a solução apresentada é por demais acoplada às entidades de “**loan**”. Com um pouco mais de trabalho, podemos deixar a nova *engine* também parametrizável, de forma a poder ser utilizada para renderizar outras seções com requisitos semelhantes (porém estruturas diferentes).

No arquivo “**config.inc.xml**” da seção “**loan**” adicione a seguinte linha:

```
<directive name="_USE_ACTOR_FILTER_" value="true" />
```

Agora, no arquivo “**list.xml**” altere a região onde estão definidos os campos do formulário de busca (tag “**search**”), adicionando ao field que mapeia o estudante que pegou o livro emprestado o atributo “**id**” com o valor “**__ACTOR__**”:

```
<search table="library.loan">
    <field type="Date" column="date" label="Date | pt_BR: Data" />
    <field type="Select" column="book" label="Book | pt_BR: Livro" link-table="library.book"
    link-column="id" link-view="name" />
    <field type="Select" column="student" label="Student | pt_BR: Estudante"
    link-table="titan._user" link-column="_id" link-view="_name" id="__ACTOR__" />
</search>
```

Por fim, no “**list.prepare.php**” do seu componente altere o código para:

```
<?
$search = new Search ('search.xml', 'list.xml');
```



```
$search->recovery ();

if ((bool) Business::singleton ()->getSection (Section::TCURRENT)->getDirective ('_USE_ACTOR_FILTER_')
&& !User::singleton ()->hasPermission ('_VIEW_ALL_'))
    $search->addBlock ('_ACTOR_', User::singleton ()->getId ());

$view = new View ('list.xml');

if (!$view->load ($search->makeWhere ()))
    throw new Exception (__ ('Unable to load data!'));

?>
```

A API do Titan será melhor detalhada no “Cookbook Master Chef”, mas podemos nos atentar a alguns elementos utilizados no código acima. O método “**getDirective**”, por exemplo, permite obter o valor das diretivas declaradas no “**config.inc.xml**” da seção, possibilitando codificar rapidamente estruturas de controle parametrizáveis. O método “**hasPermission**” é uma forma simples de perguntar ao Titan se o usuário possui determinada permissão. O método “**addBlock**” bloqueia um campo do formulário de busca com um valor específico, conforme mostrado na Figura 2.16.

Minha Biblioteca Bem vindo Administrador

Usuário Gestor ♦ Grupo(s): Administradores ♦

Pesquisar... Menu

Empréstimos: Listar Empréstimos
Biblioteca » Empréstimos

+ Criar Novo Empréstimo 🔍 Buscar Itens

Critérios de Busca Selecionados Limpar Critérios | Mudar Critérios

Estudante: Camilo Carromeu

Data	Livro	Estudante
07/22/14	Azincourt	Camilo Carromeu

1 Itens Encontrados

© 2006 - 2014 x Camilo Carromeu titanframework

Figura 2.16: Resultado do uso do método “**addBlock**” no formulário de busca.

2.3.2 Criando Tipos Derivados

Conforme já vimos, cada *field* em um XML de modelo de dados do Titan possui um atributo denominado “**type**”. O valor deste atributo, ou seja, o tipo do *field*, diz como aquele campo irá se

comportar, o formato e higienização de seu dado, as conversões necessárias para salvamento no banco de dados ou para exportação em texto puro, dentre muitas outras coisas. Os tipos do Titan obedecem, essencialmente, aos conceitos de Orientação a Objetos. Portanto, é possível estender um tipo qualquer, herdando todas as suas características, e implementar apenas as alterações que desejamos sobrescrever.

Um exemplo clássico é o tipo nativo “**Amount**”. Este tipo estende outro tipo nativo, o “**Integer**”, com o modesto objetivo de adicionar uma máscara às representações de valores inteiros de forma a separar as casas de milhar por meio do caracter ponto (“.”). Verificando o código destes dois tipos (presentes no núcleo do Titan, no diretório “**core/repos/type/**”), veremos que no tipo “**Amount**” foram sobrescritas as rotinas implementadas responsáveis pela representação visual do valor, mas todas as demais não possuem implementação própria.

Para exemplificar, vamos supor que se queira criar um campo que aceite orações (*strings*), mas cuja visualização do valor deva ser sempre em caixa alta. Para conseguir implementar este novo tipo, precisamos entender sua estrutura de código. Todos os tipos do Titan possuem uma classe com um nome único. Esta classe pode ter diversos métodos (veja a interface de tipos em “**core/class/Type.php**”), mas no âmbito deste livro considere implementar os seguintes:

- **__construct:** É essencial ao tipo ter um construtor (é o único método obrigatório). Este método recebe o nome da tabela e um vetor com todos os atributos (e seus valores) da *tag field* correspondente no XML. Ele deve sempre chamar, logo no início, o construtor da classe herdada. Por exemplo, veja abaixo a classe “**Amount**” com a implementação mínima necessária para a criação de um novo tipo no Titan:

```
<?
class Amount extends Integer
{
    public function __construct ($table, $field)
    {
        parent::__construct ($table, $field);
    }
}
?>
```

- **isValid:** É um método chamado quando o formulário é submetido. Nele o valor é testado e,

caso seja um valor aceito, o método retorna *'true'*, caso contrário retorna *'false'*. Por exemplo, veja abaixo a implementação deste método para o tipo “**Cpf**”:

```
public function isValid ()
{
    if ($this->isEmpty ())
        return TRUE;

    $cpf = (string) $this->getValue ();

    if (strlen ($cpf) != 11 || !is_numeric ($cpf))
        return FALSE;

    if ($cpf == str_pad ('', 11, $cpf [0]))
        return FALSE;

    $soma = 0;

    for ($i = 0 ; $i < 9 ; $i++)
        $soma += ((10 - $i) * $cpf [$i]);

    $d1 = ($soma % 11);

    $d1 = $d1 < 2 ? 0 : 11 - $d1;

    if ($d1 != (int) $cpf [9])
        return FALSE;

    $soma = 0;

    for ($i = 0 ; $i < 10 ; $i++)
        $soma += ((11 - $i) * $cpf [$i]);

    $d2 = ($soma % 11);

    $d2 = $d2 < 2 ? 0 : 11 - $d2;

    if ($d2 != (int) $cpf [10])
        return FALSE;

    return TRUE;
}
```

- **isEmpty:** Quando o campo (*field*) é de preenchimento obrigatório (possui o atributo “required” com valor *'true'*), este método será chamado no momento da submissão do formulário e, caso retorne *'true'*, o Titan emite um erro de campo obrigatório não-preenchido. Por exemplo, veja a implementação dele no tipo “**Cpf**”:

```
public function isEmpty ()
{
    return is_null ($this->getValue ()) ||
        trim ($this->getValue ()) == '' ||
        !(int) $this->getValue ();
}
```

```
}
```

Além dos métodos da classe, também devemos considerar a implementação dos *adapters* do tipo que, por uma decisão de projeto, são implementados em arquivos independentes, dentro da pasta do tipo. São eles: “**fromApi.php**”, “**fromDb.php**”, “**fromForm.php**”, “**fromLdap.php**”, “**fromSearch.php**”, “**toApi.php**”, “**toBind.php**” (para quando se utiliza *prepared statements*²² no banco de dados), “**toConstraint.php**”, “**toDbMaker.php**” (para engenharia reversa, ou seja, gerar a estrutura do banco de dados a partir da aplicação), “**toForm.php**”, “**toGroupBy.php**”, “**toHtml.php**”, “**toLdap.php**”, “**toList.php**”, “**toOrder.php**”, “**toSearch.php**”, “**toSql.php**”, “**toSql.SQLite.php**”, “**toText.php**”, “**toValue.php**”, “**toWhere.php**” e “**toWhere.SQLite.php**”. Também é possível criar código JavaScript próprio para o tipo (arquivo “**_js.php**”) e métodos que podem ser chamadas assincronamente por estes JavaScript (arquivo “**_ajax.php**”).

Tendo em vista esta breve descrição da estrutura dos tipos do Titan, vamos exemplificar o uso desta API criando o tipo “**UpperText**”, descrito acima. Para tal, primeiramente iremos criar no diretório “**local**” (na raiz da instância) a pasta “**type**” e, dentro dela, criaremos a pasta do nosso tipo no formato “**meuSistema.MeuTipo**” (por exemplo, “**library.UpperText**”). Dentro desta pasta criaremos um arquivo denominado “**UpperText.php**” com o seguinte conteúdo:

```
<?php
class UpperText extends Phrase
{
    public function __construct ($table, $field)
    {
        parent::__construct ($table, $field);
    }
}
```

Como a única alteração será na apresentação do valor do campo, precisamos sobrescrever apenas os adapters que atuam na camada de visualização. Os mais icônicos deles são o “**toHtml.php**” e o “**toText.php**”. Crie portanto estes arquivos em sua pasta e adicione o seguinte

²² <http://php.net/manual/en/pdo.prepared-statements.php>

código em ambos:

```
<?php  
return strtoupper ($field->getValue ());
```

Pronto! Seu novo tipo já está criado. Para ativá-lo na instância, edite o arquivo “**configure/type.xml**” adicionando uma entrada para este novo tipo:

```
<?xml version="1.0" encoding="UTF-8"?>  
<type-mapping>  
  <type name="UpperText" path="local/type/library.UpperType/" />  
</type-mapping>
```

Também é necessário garantir que a tag “**type**” esteja corretamente declarada no arquivo “**configure/titan.xml**” (normalmente ela existe, mas está comentada em instâncias recém criadas):

```
<type  
  xml-path="configure/type.xml"  
>
```

Tudo pronto! Agora basta referenciar seu novo tipo nos *fields* dos XMLs que mapeiam o modelos de dados:

```
<?xml version="1.0" encoding="UTF-8"?>  
<form table="tabela" primary="id">  
  <go-to flag="success" action="[default]" />  
  <go-to flag="fail" action="[same]" />  
  <field  
    type="UpperText"  
    column="coluna"  
    label="Caixa Alta"  
    required="true"  
    help="Este texto irá parecer em caixa alta."  
  />  
</form>
```

2.4 Publicando a Aplicação

Uma vez que a aplicação tenha chegado a um estado maduro, podemos publicá-la em um servidor Web. Neste momento é importante atentar-se aos requisitos (configurações e pacotes) necessários à boa execução do Titan Framework. Até então não foi necessário nos preocuparmos com isso pois a TitanBox, ou seja, a máquina virtual que implementa o ambiente de desenvolvimento, já têm todas as configurações e pacotes recomendados.

Uma vez que o sistema esteja rodando em produção, pode-se utilizar o sistema e *auto-deploy* do Titan para mantê-lo atualizado.

2.4.1 Configurando um Ambiente de Produção

O Titan está homologado para rodar em servidores **Linux Debian Wheezy 64 Bits**, portanto nesta seção considera-se que está sendo criado um ambiente de produção com este sistema operacional. Inicie o processo instalando os pacotes básicos no seu servidor:

```
aptitude update
aptitude install vim build-essential bzip2 subversion apache2 mailutils postfix php5 php5-pgsql php5-gd
php5-imagick php5-sqlite php5-cli php5-curl php5-mcrypt php5-ldap php-pear php5-dev php-apc xpdf-utils
antiword openjdk-7-jre
```

Habilite o suporte aos idiomas utilizados na instância. Supondo que ela utilize português (brasileiro), inglês e espanhol, para que o suporte a estes *locales* funcione corretamente no sistema é preciso ativá-los:

```
dpkg-reconfigure locales
```

Em seguida, da lista que aparece, selecione as opções:

```
en_US UTF-8
es_ES UTF-8
pt_BR UTF-8
```

Para instalar o **PostgreSQL**, execute o comando:

```
aptitude install postgresql-9.1
```

Para configurá-lo:

```
vim /etc/postgresql/9.1/main/pg_hba.conf
```

Troque as linhas:

```
# "local" is for Unix domain socket connections only
local all all peer
```

Para:

```
# "local" is for Unix domain socket connections only
local all all trust
```

Isso fará com que aplicações rodando localmente e acessando o **PostgreSQL** por *sockets* Unix não necessitem de autenticação (senha). Para as instâncias do Titan esta característica é importante para ativar o “***backup on demand***²³”.

```
/etc/init.d/postgresql restart
```

Para configurar o **PHP**, dê os comandos:

```
mkdir -p /var/www/file/tmp
vim /etc/php5/apache2/php.ini
```

E edite as diretivas abaixo com os respectivos valores:

```
open_basedir = /var/www
post_max_size = 64M
upload_tmp_dir = /var/www/file/tmp
upload_max_filesize = 64M
session.save_path = "/var/www/file/tmp"
date.timezone = America/Campo_Grande
```

²³ <https://groups.google.com/d/topic/titan-framework/ZDopSu-FKFQ/discussion>


```
sendmail_path = "/usr/sbin/sendmail -t -i"
```

Configure o **APC**, adicionando, ao final do “**php.ini**”, as seguintes linhas:

```
[apc]
apc.enabled=1
apc.shm_segments=1
apc.shm_size=32
apc.ttl=7200
apc.user_ttl=7200
apc.num_files_hint=1024
apc.mmap_file_mask=/var/www/file/tmp/apc.XXXXXX
apc.enable_cli=1
```

Para configurar o **Apache**, adicione ao final do arquivo “**/etc/apache2/apache2.conf**” as linhas:

```
ServerTokens ProductOnly
ServerSignature Off
```

Elas inibem a exibição de informações do servidor em páginas de erro do **Apache**. Na configuração do site no **Apache** não se esquecer de remover a listagem de diretórios:

```
Options Indexes FollowSymLinks MultiViews
```

Ative o “**mod_rewrite**”:

```
a2enmod rewrite
```

Edite a configuração do seu site no **Apache** (por exemplo, “**/etc/apache2/sites-available/default**”) alterando as linhas:

```
Options FollowSymLinks MultiViews
AllowOverride None all
Order allow,deny
allow from all
```

Configure o site no Apache e, por fim, o reinicie:

```
/etc/init.d/apache2 restart
```

2.4.2 Auto-Deploy e Database Migrations

O Titan possui um processo de *auto-deploy* para as instâncias. Este processo funciona delegando ao *scheduler job* do sistema operacional a tarefa de fazer a chamada a um *bootstrap* que dispara a verificação da work copy em produção com as revisões enviadas pela equipe de desenvolvimento. Caso haja um novo *release*, este é colocado em produção. O procedimento também implementa, em parte, o conceito de *database migrations* do *Ruby on Rails*²⁴, que é a gestão e rastreabilidade de mudanças incrementais no banco de dados da aplicação.

Para que o procedimento de *auto-deploy* funcione corretamente, existe na raiz da instância a seguinte estrutura de diretórios:

```
MyApp/  
  configure/  
  section/  
  ...  
  update/  
    app/  
      production.txt  
      test.txt  
      ...  
    db/  
      20130220123401.sql  
      20130225021401.sql  
      20130227234701.sql  
      ...  
  blacklist.txt
```

Os arquivos “**TXT**” dentro da pasta “**update/app**” relatarão quais arquivos devem ser atualizados naquela revisão. São chamados de “**arquivos de caminho**” (“*file of paths*”). O nome do arquivo define o ambiente em que a atualização será aplicada.

Na pasta “**update/db**” ficam as alterações a serem aplicadas no banco de dados. As

²⁴ <http://guias.rubyonrails.com.br/migrations.html>

alterações devem sempre ser colocadas no formato “**YYYYMMDDHHUUNN.sql**”, onde: 'YYYY' é o **ano**, 'MM' é o **mês**, 'DD' é o **dia**, 'HH' é a **hora**, 'UU' é o **minuto** e 'NN' é um **número sequencial** iniciado em 01 (para cada minuto). Estes valores são referentes ao momento em que o arquivo é criado. O nome do arquivo é a versão do banco de dados.

Além da criação desta estrutura você precisa inserir configurações adicionais no “**configure/titan.xml**” da sua instância:

```
<update
  environment="test"
  svn-login="update"
  svn-password=""
  svn-users="camilo"
  backup="true"
  file-mode="664"
  dir-mode="775"
  owner="root"
  group="staff"
/>
```

O atributo “**environment**” diz qual é o ambiente em que a instância está executando. É fundamental para que o *script* saiba qual *file of paths* deverá considerar. Os atributos “**svn-login**” e “**svn-password**” devem ser configurados com um usuário e senha do repositório SVN com o código da instância. O atributo “**svn-users**” define quais usuários possuem *commits* que podem ser atualizados, ou seja, caso seja commitado um *file of paths* por um usuário não especificado nesta diretiva, ele será desconsiderado no momento do *update*. O atributo “**backup**” diz se deverá ou não ser realizado o *backup* do banco de dados antes que sejam aplicadas alterações. Repare que o *backup* apenas será efetuado se houverem alterações a serem aplicadas no banco de dados.

Atenção! Caso não seja explicitamente declarado o valor '*false*' nesta diretiva, o *script* de *auto-deploy* sempre tentará fazer o *backup*, ou seja, o valor padrão deste atributo é '*true*'. A pasta de *backup* será a mesma utilizada pela funcionalidade “**backup on demand**²⁵”. Atente-se para o fato de que esta funcionalidade considera um período de validade máxima para os arquivos gerados, que serão automaticamente apagados ao expirar este período visando preservar o espaço físico do servidor.

²⁵ <https://groups.google.com/d/topic/titan-framework/ZDopSu-FKFQ/discussion>

As demais configurações (“**file-mode**”, “**dir-mode**”, “**owner**” e “**group**”) devem ser setadas apenas caso a instância não esteja em um ambiente Debian. Referem-se às permissões que os arquivos atualizados receberão.

A primeira vez que o procedimento é executado ele cria automaticamente uma tabela denominada “**__version**” no *schema* do Titan no banco de dados. Esta tabela irá controlar as versões do banco de dados, quem aplicou cada versão e quando ela foi aplicada.

De forma geral, o procedimento de *auto-deploy* funciona da seguinte forma:

1. Primeiro ele captura a versão atual da *work copy* no servidor e a *head revision* do *file of paths* para o ambiente no repositório SVN;
2. Por meio de um laço ele atualiza o *file of paths* para a versão imediatamente mais recente (revisão atual + 1);
3. Para cada iteração do laço, ele atualiza a pasta “**update/db**” para a mesma revisão e verifica qual a última versão do banco de dados (consultando a tabela “**__version**”);
4. Caso haja alguma alteração a ser aplicada no banco de dados, ele efetua *backup*;
5. Os arquivos “**SQL**” da pasta “**update/db**” que serão aplicados ao banco de dados são então atualizados para a *head revision*;
6. Então, ele aplica as alterações no banco de dados. Caso ocorra erro, é realizado um *rollback* no banco e o *file of paths* é atualizado para a última revisão corretamente aplicada;
7. Em caso de sucesso os arquivos (*paths*) listados no *file of paths* são atualizados para a revisão da iteração; e
8. Por fim, não havendo problemas esta revisão passa a ser a última revisão estável e o laço avança para a próxima iteração.

Desta forma, mesmo um sistema que esteja sendo instanciado no servidor a partir de um *backup* antigo terá todas as revisões metodicamente aplicadas recursivamente.

Vale lembrar que, independente da quantidade de iterações do laço que possuam alterações

a serem aplicadas no banco de dados, o procedimento irá gerar, por execução, apenas um *backup* inicial (e somente se houver alterações a serem aplicadas).

Repare também na existência do arquivo “**update/blacklist.txt**”. Neste arquivo podem ser colocadas revisões que devem ser desconsideradas. Assim, o *script* sempre atualiza este arquivo para a *head revision* e consulta seu conteúdo, permitindo manipular as revisões que serão de fato aplicadas e remover revisões que estejam causando erros recorrentes no procedimento (obrigando o *rollback*).

Para que o procedimento funcione, é necessário que esteja executando em um servidor *NIX, sendo que está homologado especificamente para Debian. Será necessário também ter instalado no PHP a biblioteca SVN do PECL:

```
su -  
aptitude update  
aptitude install build-essential php-pear php5-dev libsvn-dev  
pecl update-channels  
pecl install svn  
echo "extension=svn.so" > /etc/php5/conf.d/svn.ini  
/etc/init.d/apache2 restart
```

Para executar o script basta chamá-lo passando como parâmetro todas as instâncias que deseja atualizar (sem limite de instâncias). O script em si (que deverá ser chamado) está localizado na pasta “**update**” do *core* do Titan. Por exemplo:

```
php /var/www/titan/update/update.php /var/www/portal/manager/ /var/www/pandora/
```

No exemplo acima o *script* irá atualizar em sequência as duas instâncias passadas. Lembrando que cada uma delas deverá ter a pasta “**update**” (como a estrutura de arquivos e diretórios descrita) e deverá ter a *tag* “**update**” em seu “**configure/titan.xml**”.

Conhecendo o funcionamento do procedimento, basta agora inserí-lo no *scheduler job* do sistema operacional. No caso do Debian, edite o arquivo “**/etc/crontab**” inserindo a linha abaixo:

```
*/5 * * * * root /usr/bin/php /var/www/[caminho para oTitan]/update/update.php /var/www/[caminho para a primeira instância] /var/www/[caminho para a segunda instância] > [arquivo com log de saída]
```

Neste caso, o procedimento de *auto-deploy* irá executar a cada 5 (cinco) minutos. Altere o número '5' para aumentar o período de tempo:

```
*/20 * * * * root /usr/bin/php /var/www/titan/update/update.php /var/www/portal/manager/  
/var/www/pandora/ > /var/log/titan-auto-update.log
```

Ou seja, na configuração acima as duas instâncias do nosso exemplo inicial serão atualizadas a cada 20 minutos e o LOG será jogado no arquivo “**/var/log/titan-auto-update.log**”. Lembre-se de reiniciar o **CRON** após editar o “**/etc/crontab**”:

```
/etc/init.d/cron restart
```

Quando a instância é atualizada com sucesso, ao invés de jogar no arquivo de *log* as informações da atualização, estas são enviadas para o e-mail dos usuários que pertencem a grupos com permissão de administrador na instância.