



Apostila de desenvolvimento Web
CodeIgniter + RedBeanPHP
v1.1b

Prof: Alan Klinger

O que é o CodeIgniter?

O CodeIgniter é um framework para construção de aplicações Web em PHP. Um framework é um conjunto de bibliotecas já organizadas em uma estrutura que deve ser seguida pelo desenvolvedor. Ela facilita o desenvolvimento, pois, existem várias bibliotecas para ajudar em funcionalidades comuns, como por exemplo: sessões, acesso ao banco de dados, upload de arquivos, entre outros.

Mas além disso o CodeIgniter é um framework MVC. Ou seja, ele trabalha no padrão MVC, que é um padrão já utilizado em diversas aplicações. A ideia desse padrão é organizar o código em 3 partes, Model, View e Controller (Modelos, Visões e Controladores).

Essa organização é útil para que não se misture código de acesso a banco de dados com código HTML, por exemplo:

```
<?php
$sql = "select * from usuarios";
$rs = mysqli_query($sql);
...
?>

<table>
    <?php
    for (...) {
        print "<td>{$rw['nome']}</td>";
    }
</table>
```

Um código para imprimir tudo da tabela de usuário seria algo semelhante a isso, foram omitidas algumas linhas, veja que temos código PHP, SQL e HTML, tudo em um único arquivo. E se precisássemos imprimir todos os usuários em outro lugar além desse? Teríamos esse mesmo código, todo de novo nesse outro lugar. Caso seja necessário alterar o SQL, seria preciso alterar em todos os lugares.

O padrão MVC tenta organizar isso, de modo que somente o arquivo do Model poderá acessar o banco de dados, os arquivos de View serão responsáveis apenas pela construção do HTML.

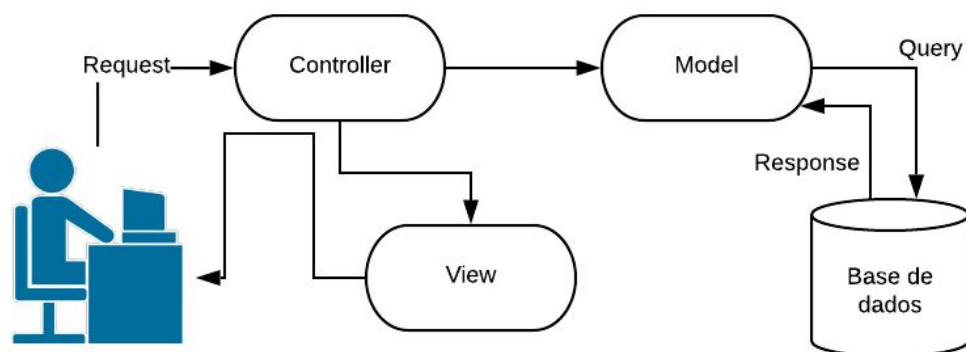


Figura 1 - Padrão MVC

Imagine que o usuário quer ver todos os usuários cadastrados. Como apresentado na figura 1, toda requisição que o usuário fizer em um site com o padrão MVC irá para o Controller, que terá como papel, decidir o que será acessado. No caso ele irá acessar o Model responsável pela tabela de usuários que irá retornar os dados solicitados e esses dados serão enviados para a View de listagem de usuários.

Sendo assim, sempre existirá um Model para cada tabela, bem como uma View para cada tabela. Porém, isso não é regra, o programador pode implementar novos meios de reaproveitar o código.

O que é o RedBeanPHP?

O RedBeanPHP é uma biblioteca para acesso ao banco de dados. O CodeIgniter já possui uma biblioteca padrão, porém a biblioteca nativa não tem tantas funcionalidades quanto o RedBean. No caso, o CodeIgniter permite que sejam instalados plugins, assim foi criado um projeto com o CodeIgniter com o plugin do RedBean.

Uma das funcionalidades que o RedBean possui é o de criação e alteração de tabelas sem a necessidade de interferência do programador. Por exemplo:

Se eu criar e executar o código PHP:

```
$livro = R::dispense( 'livros' );  
$livro->titulo = 'Livro 1';  
$livro->paginas = 100;  
$id = R::store( $livro );
```

O próprio PHP, através do RedBean, irá criar a tabela **“livros”**, no banco de dados, já com os campos, id, titulo e paginas, e irá inserir o “Livro 1” que contém 100 páginas.

Caso fossemos usar a biblioteca padrão do CodeIgniter, precisaríamos anteriormente criar a tabela no banco de dados para depois inserir dados. Lembrando que se eu quiser criar mais um campo, basta adicionar mais uma linha:

```
$livro->preco = 98.50;  
$id = R::store( $livro );
```

Sqlite e Mysql

Tanto o Sqlite quanto o Mysql são banco de dados, porém o Sqlite é bem mais simples, possuindo menos funcionalidades, e trabalha com apenas um arquivo. Em outros termos, é um banco de dados para testes, sugerido para quem está criando novas aplicações.

O principal recurso do Sqlite, é que ele trabalha com um único arquivo, basta copiar esse arquivo junto com o código PHP que o banco de dados será copiado junto. Ou seja, basta copiar e colar o sistema em outra máquina e o site já estará em funcionamento. Sem a necessidade de restaurar backups ou instalar qualquer outro software.

Caso você precise visualizar o banco de dados SQLite baixe o programa: <https://sqlitebrowser.org/dl/> extraia e execute o: “DB Browser for SQLite.exe”

O banco de dados sempre estará na pasta **application/db/** do projeto que nós usaremos.

O exemplo

Essa apostila irá trabalhar com dois exemplos de uma aplicação já construída com o CodeIgniter + RedBean. A primeira aplicação será mais simples, para que você possa ser introduzido ao CodeIgniter, a segunda aplicação já contará com diversas funcionalidades que serão explicadas na apostila para que cada aluno possa modificar o projeto de acordo com as suas necessidades.

A primeira aplicação, o CRUD

Um CRUD é uma página que consegue salvar, alterar, excluir e listar alguma coisa. É a funcionalidade básica de um sistema.

[Faça o download do exemplo do primeiro crud.](https://github.com/klingerkrieg/codeIgniterSqlite/tree/master)
(<https://github.com/klingerkrieg/codeIgniterSqlite/tree/master>)

Extraia o arquivo na pasta htdocs do xampp. Normalmente em c:/xampp/htdocs, como mostra a figura 2:

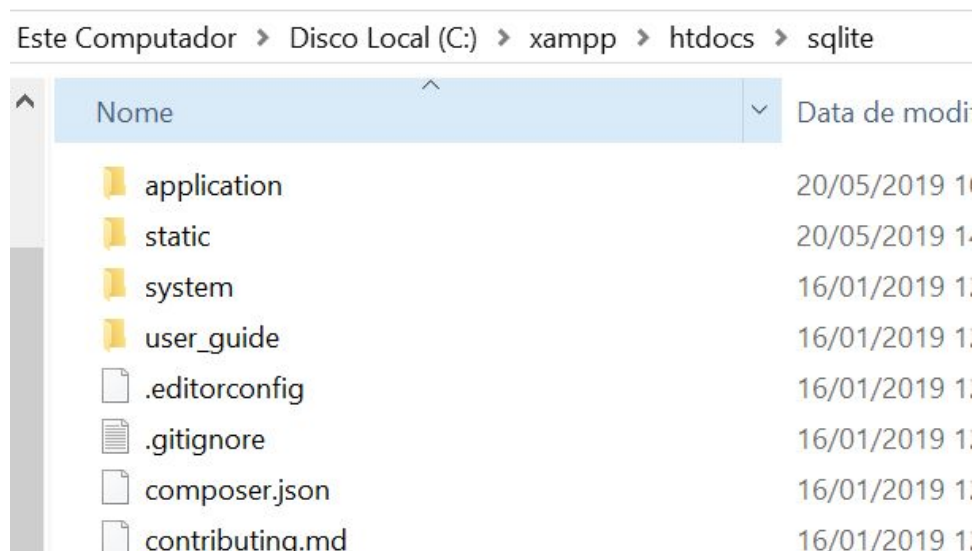
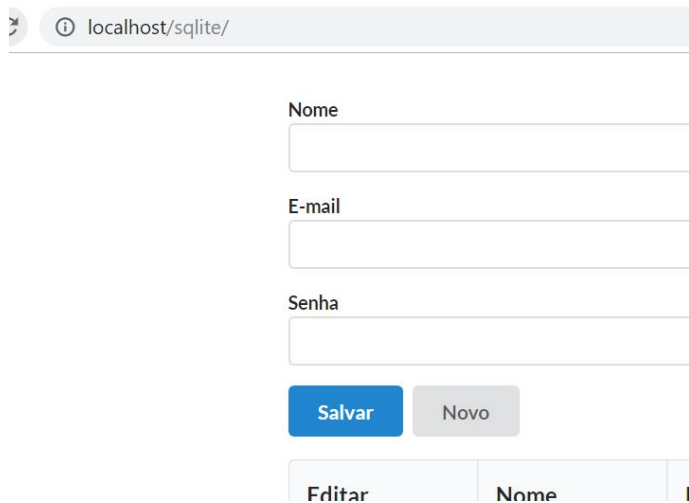


Figura 2 - Pasta do projeto

Os arquivos devem ficar como na imagem, todos os arquivos foram extraídos para uma pasta chamada **sqlite** que ficará dentro de htdocs. Abra o xampp e inicie somente o Apache, o mysql não será utilizado.

Nesse primeiro momento é importante que você mantenha o nome da pasta como **sqlite**, posteriormente, falaremos do config.php onde você poderá alterar o nome da pasta.

Acesse o sistema pelo navegador através do endereço: <http://localhost/sqlite> . O sistema deverá aparecer de imediato, como mostra a figura 3.



localhost/sqlite/

Nome

E-mail

Senha

Salvar Novo

Editar	Nome	F
--------	------	---

Figura 3 - Tela do projeto

Teste a aplicação, para verificar se ela está salvando dados.

Controller

Para iniciar a explicação iremos começar pelo início do fluxo, o Controller, que é o arquivo que recebe a requisição do usuário e chama o Model e a View responsável.

Abra o arquivo `sqlite/application/controllers/Usuarios.php`

Nesse arquivo teremos:

```
class Usuarios extends CI_Controller { //<----- O nome da classe,  
    // sempre deve ser igual ao nome do arquivo  
  
    //Construtor da classe  
    public function __construct(){  
        parent::__construct(); //Sempre deverá ter essa linha,  
        //Sempre que iniciar essa classe,  
        //ele carregará o model de Usuario  
        $this->load->model("Usuario_model");  
    }  
}
```

Em seguida temos o método **index**, para entendermos esse método é preciso saber três coisas:

A) Sempre que existir um método **index**, ele será o principal método do Controller, caso o usuário digite o endereço: <http://localhost/sqlite/index.php/usuarios> o Controller **usuarios** irá executar o método **index**.

Se você criar um Controller com outro nome, por exemplo, **alunos**, o endereço seria: <http://localhost/sqlite/index.php/alunos> e com esse endereço ele iria acessar o **index** de alunos.

Também é possível digitar o caminho completo: <http://localhost/sqlite/index.php/alunos/index>.

Caso eu queira acessar outro método eu sou obrigado a colocar o endereço completo: <http://localhost/sqlite/index.php/usuarios/salvar>

O endereço sempre será formado por:

<http://localhost/sqlite/index.php/usuarios/index>

[http://\[endereco\]/\[pastaprincipal\]/index.php/\[controller\]/\[metodo\]](http://[endereco]/[pastaprincipal]/index.php/[controller]/[metodo])

Esse padrão é gerado pelo MVC em qualquer framework que siga sua regra. O arquivo index.php na url é uma necessidade do CodeIgniter, porém ele pode ser omitido com uma configuração, isso será abordado na segunda parte da apostila.

Esse padrão tende a proteger os arquivos do PHP, porque o cliente jamais conseguirá acessar um model ou uma view sem passar pelo controller. Por exemplo, ele não consegue digitar: http://localhost/sqlite/application/models/Usuario_model.php e acessar.

B) Esse exemplo já conta com o sistema de paginação:

Editar	braulino		Deletar
Editar	caio		Deletar
Editar	izadory		Deletar
Editar	joao		Deletar
Editar	maria		Deletar
Editar	milena		Deletar
Editar	pedro		Deletar
<div>1 2 ></div>			

Figura 4 - Paginação

Sempre será exibido no máximo 10 registros por página. Quando houver mais do que 10 registros para serem exibidos ele irá mostrar os links para as próximas páginas da tabela, de acordo com a figura 4.

C) Quando eu clicar para editar um registro, os dados da pessoa irão aparecer no formulário, para que seja possível editá-la, como mostra a figura 5:

Nome

teste

E-mail

teste@gmail.com

Senha

Salvar

Novo

Editar	Nome	E-mail
Editar	teste	teste@gmail.com

Figura 5 - Edição de registros

Sabendo disso:

```
//a função index sempre será a função principal do arquivo

//ela é aquela função que será acessada se o usuário digitar somente o
nome do controller

//http://localhost/sqlite/usuarios/

public function index($id=null) { //recebo a id do usuario via parâmetro


    //verifico se foi passada alguma página para
    //ser exibida, caso não tenha sido passada
    //irá para a página 1
    //isso porque a tabela só mostra 10 registros
    //de cada vez, sendo assim, podem existir várias páginas
    //a primeira página, mostra os primeiros 10,
    //a segunda página mostra os próximos

    if (isset($_GET['page'])){
        $page = $_GET['page'];
    } else {
        $page = 1;
    }


    //Utilizo o model para receber todos os registros do banco de dados
    //porém de forma paginada, traga somente os registros da página X
    $pag = $this->Usuario_model->pagination($page);


    //Se eu tiver enviado a id de algum individuo
    //O model usuario irá trazer os dados desse usuário
    //Para que ele seja exibido no formulário

    $dados = $this->Usuario_model->get($id);
```

```

//Chamo a VIEW
$this->load->view('usuarios', ["list"=>$pag["list"],
                                "qtd"=>$pag["qtd"],
                                "page"=>$page,
                                "dados"=>$dados]);
}

```

Para entender melhor, observe o diagrama na figura 6.

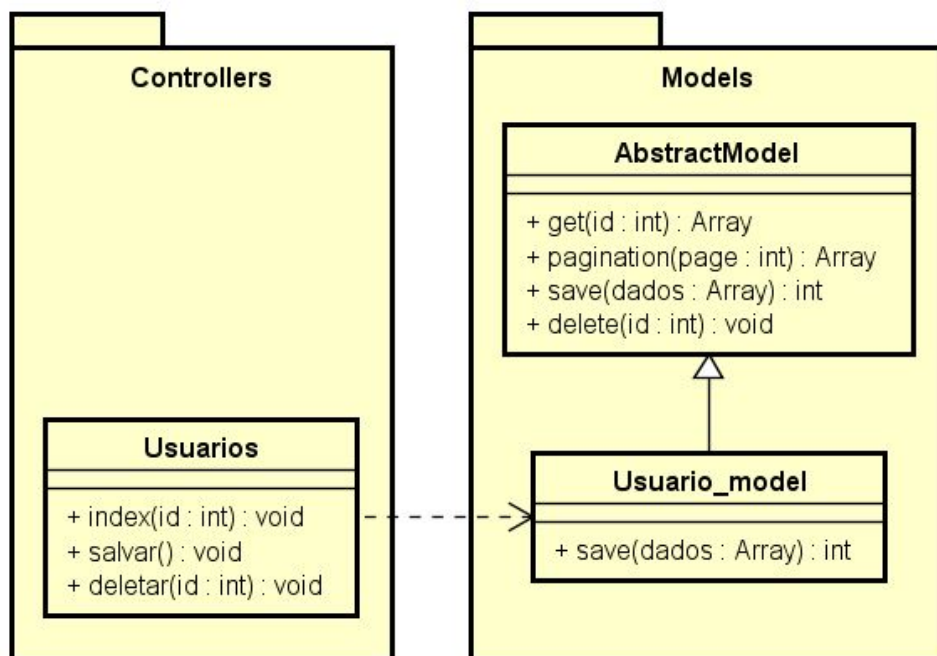


Figura 6 – Diagrama de classe, Controller Usuarios x Usuario_model

Agora abra o arquivo do Model:

sqlite/application/models/Usuario_model.php

Iremos entender a interação entre o Controller e o Model. Primeiramente, vimos que ele chama o método **paginacao** e **get** do

Model de **usuarios**, porém quando abrimos o Model esses métodos não existem, isso porque eles estão na classe mãe, **AbstractModel**. Todos os métodos mais comuns estão dentro de **AbstractModel**, dessa forma não é preciso reescrevê-los dentro de **Usuario_model**, isso só será necessário se o comportamento do método precisar mudar.

```
class Usuario_model extends AbstractModel {  
    public $table = "usuarios";  
    public $fields = ["nome","email","senha"];
```

Observe que dentro do Model começamos com **\$table** e **\$fields**, onde deverá ter, respectivamente o nome da tabela e o nome dos campos da tabela. Note que não são definidos os tipos dos campos, o RedBean irá defini-los automaticamente. Porém eles podem ser redefinidos manualmente, diretamente no banco de dados.

Dessa forma, o método **pagination**, que é chamado primeiro pelo Controller encontra-se dentro do **AbstractModel**:

```
//recebe o número da página que será exibida, inicia na 1  
public function pagination($page){  
  
    //Quantidade de itens por pagina  
    $max_items_per_page = 10;  
    $loc = ($page-1) * $max_items_per_page;  
  
    //Seleciona todos os dados, ordenando pelo primeiro campo da tabela  
    $list = R::findAll($this->table , " ORDER BY " . $this->fields[0]  
        . " LIMIT $loc,$max_items_per_page " );
```

```

    //Recupera a quantidade total de itens na tabela
    $qtd = R::count($this->table);

    //Retorna um array com a list, registros na página,
    //e a quantidade total de itens.
    return ["list"=>$list,"qtd"=>$qtd];
}

```

Observe que esse método retorna um array com duas coisas, os registros que serão exibidos, no máximo 10 registros e a quantidade total de registros, isso para que posteriormente sejam exibidos os links para as próximas páginas.

O segundo método chamado é o **get**.

```

//recebo a id do elemento que quero abrir
public function get($id = null){
    if ($id == null){
        //se a id for nula, retorna um array vazio
        return [];
    } else {
        //caso contrário, busca no banco
        //aquele elemento da tabela que tem aquela id
        return R::load($this->table,$id);
    }
}
}

```

Note que o RedBean é acessado sempre que utilizamos “R::”, o próprio RedBean já possui vários métodos que serão explicados ao

decorrer da apostila. O **R::load**, retorna uma linha do banco de dados, para isso, você precisa passar a tabela e a id do registro.

No método anterior foram utilizados, **R::findAll** e **R::count**, que trazem todos os registros da tabela, de acordo com o filtro e conta os registros da tabela, de acordo com o filtro. Caso haja dúvidas na utilização desses métodos procure a documentação oficial do RedBean, que já conta com vários exemplos: <https://redbeanphp.com/index.php>

Uma vez que estamos no Controller e já conversamos com o Model, precisamos chamar a View que irá exibir os dados.

```
//Chamo a VIEW
$this->load->view('usuarios', ["list"=>$pag["list"],
                                "qtd"=>$pag["qtd"],
                                "page"=>$page,
                                "dados"=>$dados]);
```

Essa é a linha responsável por chamar a View, no caso ela está chamando a view, usuarios.php e passando os dados ao lado.

Os dados passados para a View chegarão lá como variáveis, no caso **\$list** é um array com os usuários que serão listados, **\$qtd** é a

quantidade de usuários, **\$page** é o número da página onde estou e **\$dados** são os dados do usuário que está sendo editado:

```
[ "list"=>$pag[ "list" ],  
  "qtd"=>$pag[ "qtd" ],  
  "page"=>$page,  
  "dados"=>$dados ]
```

```
$list <- array  
$qtd <- número  
$page <- número  
$dados <- array
```

Abra essa view em: `sqlite/application/views/usuarios.php`

Observe que na primeira linha dessa view, ela inclui outro arquivo, o header.

```
<?php include 'layout/header.php' ?>
```

No **header.php** nós temos um código pela metade, isso porque a ideia é que no header esteja presente os includes para todos os arquivos **.js** e **.css** que existirão em todas as páginas, bem como o **bottom.php** terá o fechamento das tags que foram abertas aqui.

```
<html>  
  
<head>  
  
<link rel="stylesheet" type="text/css"  
href="<?=base_url()>static/semantic/semantic.css" />  
  
<link rel="stylesheet" type="text/css" href="<?=base_url()>static/estilo.css"  
/>  
  
<script src="<?=base_url()>static/semantic/semantic.js"></script>  
  
</head>  
  
<body>
```

```
<div class="ui text container">
```

Observe que os arquivos **.js** e **.css** estão dentro de uma pasta chamada **static**, que fica fora do **application**. Por isso deve ser usado o **base_url()** e não o **site_url()**:

base_url()	http://localhost/sqlite	Usado para coisas fora do application.
site_url()	http://localhost/sqlite/index.php	Usado para controllers.

Obrigatoriamente, todos os arquivos **.js**, **.css**, **.pdf**, imagens, devem ficar na pasta **static**, isso porque eles não precisam ser protegidos, podem ser acessados sem login.

Com isso podemos ter os arquivos das views mais simples, apenas com o conteúdo da página, não precisamos ficar incluindo scripts e css em todas as views, já que eles já estarão incluídos no **header.php**. Nesses exemplos estaremos usando o Semantic CSS, <https://semantic-ui.com/>, para dar um visual mais bonito ao formulário, porém você pode trocar por um framework de sua preferência, Bootstrap, Materialize, etc. No final teremos algo como:

views/layout/header.php	Inclusões de arquivos .css e .php e outras coisas que aparecerão em todas as páginas, como menus.
views/usuarios.php	Conteúdo da página
views/layout/bottom.php	Fechamento para o que foi aberto no header.php

O PHP pegará todos esses arquivos, irá juntar em uma única página e enviará para o navegador do usuário como se fosse um arquivo só.

Voltando à View **usuarios.php** observe o action do form:


```
<form class="ui form column stackable grid"
action="<?=site_url()?>/usuarios/salvar" method="post">
```

Foi utilizada uma função do PHP dentro do action, isso porque eu não posso colocar o endereço fixo: <http://localhost/site/usuarios>. Mas por quê? Porque esse endereço pode mudar, basta modificar o nome da pasta que o endereço muda, então utilizamos a função **site_url()** porque ela retorna parte do endereço correto, faltando apenas indicar o controller e o método: <http://localhost/site/index.php> faltando [/usuarios/salvar](#) .

Seguindo mais adiante, você vai perceber que em todos os values é utilizado a função **val()**, essa função está definida dentro do arquivo **sqlite/application/libraries/Rb.php**. Ela trabalha da seguinte forma, você passa um array e a chave da casa que você quer pegar do array. Caso a chave exista ele retorna o valor que você quer, caso não exista ele retorna o valor em branco. Evitando erros de chave inexistente.

```
<input type="hidden" name="id" value="<?=val($dados, 'id')?>">
```

Já nos botões, perceba que foi utilizado o botão do tipo “submit” para enviar o formulário, ou seja. Não foram feitas validações no Javascript. No próximo exemplo essas validações serão feitas, porém somente no lado servidor.

O botão de “novo”, é na verdade um link, para a própria página.

```
<div class="field">
```

```
<button class="ui blue button" type="submit">Salvar</button>
```

```
<a class="ui button" type="button" href="<?=site_url()?>/usuarios">Novo</a>
</div>
```

Logo em seguida temos a listagem dos dados cadastrados, a listagem é necessária para que possamos ter a funcionalidade de abrir algum registro para edição ou excluir algum registro. Lembrando que essa listagem é paginada, ou seja, só aparecerão 10 registros por página, assim que você cadastrar mais que 10, a paginação aparecerá no final da tabela:

```
<table class="ui celled table">
  <thead>
    <tr>
      <th>Editar</th>
      <th>Nome</th>
      <th>E-mail</th>
      <th>Deletar</th>
    </tr>
  </thead>
  <tbody>
```

Após iniciarmos a tabela com seu cabeçalho temos um foreach para imprimir os dados.

```
<?php
foreach($list as $ln){
  print "<tr>";
  print "<td><a href='".site_url()."/usuarios/index/{$ln->id}'> Editar
</a></td>";
```

Construímos o link para edição do registro. Estamos apontando para o método **index** do controller **usuarios**, passando a **id** do

usuario, ou seja, o link vai ser algo como:

<http://localhost/sqlite/index.php/usuarios/index/1> , vá no método **index** do **usuarios**, e perceba que ele recebe um parâmetro, **\$id = null**, que significa que se você passar uma id na URL ele preenche essa variável, caso você não passe nenhuma id, essa variável será **null**.

```
print "<td>{$ln->nome}</td>";  
print "<td>{$ln->email}</td>";
```

A lógica para a criação do link do deletar é a mesma, só muda o método para o qual será enviado, será o método **deletar** do controller **usuarios**.

```
print "<td><a href='".site_url()."/usuarios/deletar/{$ln->id}'> Deletar  
</a></td>";  
print "</tr>";  
}
```

Após todos os dados serem impressos, há um if para controlar a paginação. A função ceil arredonda para cima, exemplo: Digamos que tenhamos 11 registros, $11/10 = 1.1$, arredonda 1.1 para cima = 2. Teremos 2 páginas para 11 registros. A primeira página mostra os 10 primeiros, a segunda mostra o último.

```
#paginacao, se houver mais de uma página a paginação aparecerá  
$page_max = ceil($qtd/10);  
if ($page_max > 1):
```

Para construir a paginação, utilizamos a própria biblioteca pagination do CodeIgniter, porém ela foi estilizada para obedecer ao CSS do semantic, essa estilização foi feita no arquivo: `sqlite/application/config/pagination.php`.

```

$config['total_rows'] = $qtd;

$this->pagination->initialize($config);

?>

<tfoot>

<tr>

    <th colspan="5">

        <div class="ui right floated pagination menu">

```

Após construir o local na tabela onde a paginação irá ficar, finalmente ele imprime a paginação.

```

        <?=$this->pagination->create_links()?>

    </div>

</th>

</tr>

</tfoot>

<?php endif; ?>

</tbody>

</table>

```

Perceba que foi usado, **endif**; essa notação é para facilitar a leitura do código PHP quando está no meio do HTML. Sabemos aqui que fechamos um IF, caso contrário veríamos apenas uma chave fechando “}”, e não saberíamos o que ela está fechando, um if, um for, while? Para utilizá-la o if deve vir acompanhado com dois pontos “.”:

if(condicoes) : (...) endif;	while(condicoes) : (...) endwhile;	for (condicoes) : (...) endfor;
--	--	---

Por fim, incluímos o layout/bottom.php que fecha as tags abertas no header.php

```
<?php
include 'layout/bottom.php';
?>
```

Voltando para o Controller, salvar e deletar

Agora que já sabemos que o formulário envia para o método **salvar** do **usuarios**, vamos ver o que esse método faz:

```
public function salvar(){
    $id = $this->Usuario_model->save();
    redirect("usuarios/index/" . $id );
}
```

Basicamente, ele chama o model do usuário e manda salvar, após isso ele redireciona para o método **index** do **usuarios**, já passando a **\$id** que foi devolvida, pelo model.

O método **save** do model do usuário não existe, ele foi herdado do **AbstractModel**, o que existe é o **preSave** que é executado antes do objeto ser salvo. Esse método só foi criado em função do campo de senha, que precisa ser criptografado para ser inserido no banco, e quando a senha não for informada ele deve retirar o campo da senha do SQL para não sobrescrever a senha já existente:

```
public function preSave($obj, $data) {
    if ($obj["senha"] != ""){
        //criptografa a senha
        $obj["senha"] = sha1($obj["senha"]);
    } else {
        //caso a senha venha em branco, nao vai modificar
    }
}
```

```
    unset($obj["senha"]);  
  }  
  //a funcao preSave sempre deve retornar o $obj  
  return $obj;  
}
```

A criptografia da senha consiste em transformar a senha do usuário em algo ilegível, ex: 123456 -> 7cxb2bds8..., dessa forma ela pode ser salva no banco e ninguém conseguirá saber qual é a senha daquele usuário. Sempre que usamos o sha1 para criptografar uma senha, a resposta é sempre a mesma, ou seja, 123456 sempre vai resultar no mesmo código, 12345 irá resultar em outro. Dessa forma é possível fazer o login, desde que a senha seja criptografada novamente antes de ser pesquisada no banco de dados.

A função **unset** deleta a variável. É utilizada quando a senha vem vazia aí eu tenho que deletá-la para que ele não altere a senha no banco.

No final a função sempre deverá retornar o **\$obj** que será salvo no banco. A função **save** que foi herdada irá pegar o **\$obj**, salvar e retornar sua ID para o controller.

O deletar atua de forma semelhante, chama o método responsável do model usuario e redireciona para o index, porém, redireciona sem id, uma vez que o registro foi deletado.

```
public function deletar($id){  
    $this->Usuario_model->delete($id);  
    redirect("usuarios/index");  
}
```

Perceba que também não existe o método delete no model de usuario, ou seja, ele usa o delete do **AbstractModel**. Que é bem

simples, para deletar um item primeiro ele precisa carregá-lo, **R::load**, depois **R::trash** e pronto.

```
public function delete($id){  
    $obj = R::load($this->table,$id);  
    R::trash($obj);  
}
```

Lembrando que o **AbstractModel** é uma classe criada para adaptar o RedBean ao MVC, ele não faz parte nem do CodeIgniter nem do RedBean. Pronto, basicamente todos as funcionalidades, salvar, alterar e excluir já foram explicadas.

Configurações

Quanto ao CodeIgniter, temos mais algumas possíveis configurações no arquivo, `sqlite/application/config/config.php`

Configure a URL correta para que as funções `site_url()` e `base_url()` funcionem. Caso você mude o **nome da pasta**, é importante alterar aqui também.

```
$config['base_url'] = 'http://localhost/sqlite/';
```

É possível configurar o banco de dados em: `sqlite/application/config/database.php`

```
$db['default'] = array(  
    'dsn' => 'sqlite:'.APPPATH.'db/db.sqlite',  
    'hostname' => 'localhost',  
    'username' => '',  
    'password' => '',  
    'database' => '',
```

```
'dbdriver' => 'pdo',  
...
```

Caso queira configurar para o Mysql, faça da seguinte forma:

```
$db['default'] = array(  
    'dsn' => '',  
    'hostname' => 'localhost',  
    'username' => 'root',  
    'password' => '',  
    'database' => 'nome_do_database',  
    'dbdriver' => 'mysqli',  
    ...  
);
```

No arquivo `sqlite/application/config/autoload.php` é possível indicar, models, bibliotecas, plugins para serem carregados automaticamente em todos os controllers.

```
$autoload['libraries'] = array('database', 'rb', 'pagination');  
$autoload['helper'] = array('url');  
$autoload['model'] = array('AbstractModel');
```

No arquivo `sqlite/application/config/routes.php` nós podemos indicar qual será o controller inicial do site. Nesse caso será o controller de usuarios, ou seja, quando o cliente digitar: <http://localhost/sqlite> ele enviará para <http://localhost/sqlite/index.php/usuarios/index> , lembrando que o **index** é sempre o método principal.

```
$route['default_controller'] = 'usuarios';
```


Isso é tudo que você precisa conhecer sobre o exemplo simples, é importante entendê-lo bem antes de continuar nessa apostila.

Exemplo 2

No próximo exemplo nós temos uma aplicação bem mais completa, com tela de login, validação dos campos do formulário, mensagens de aviso, upload de arquivos, relacionamentos entre tabelas, busca e menu. Por isso que para prosseguir é preciso entender muito bem o fluxo de um CRUD simples.

[Faça o download do projeto aqui.](https://github.com/klingerkrieg/codeIgniterSqlite/tree/completo)
(<https://github.com/klingerkrieg/codeIgniterSqlite/tree/completo>)

Para iniciar, extraia para a pasta htdocs/sqlite_completo, clique em Cadastre-se, ele deverá mostrar uma mensagem com o e-mail e a senha do Admin. Futuramente você pode criar uma tela para cadastro no sistema.

Login

Antes de entender o código, entenderemos o fluxo do login na figura 7:

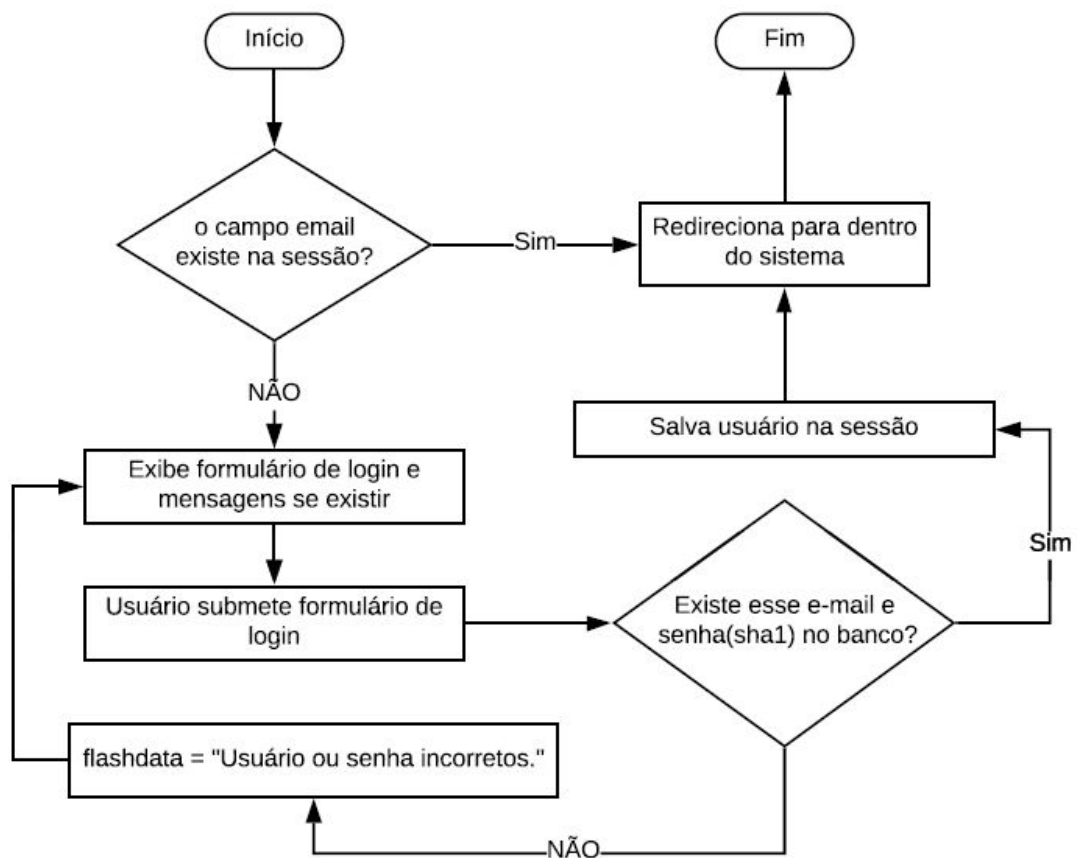


Figura 7 – Fluxo do login

Para o login, precisamos conhecer o controller/Login.php

```

public function index() {

    #verifica se o usuário já fez o login

    if (isset($_SESSION["email"])){

        redirect("usuarios/index/");

    }

    $this->load->view('login');

}
  
```

Repare que no index, a primeira coisa a ser verificada é se o usuário já está logado, se não estiver ele apresenta a tela de login. Para verificar se está logado ele apenas checa se foi criada a chave

“email” na sessão do PHP. Se já estiver logado, ele redireciona para a página pós login.

```
public function login(){

    $this->form_validation->set_rules('email', 'E-mail', 'required')

    $this->form_validation->set_rules('senha', 'Senha',

                                                'required|min_length[6]',

        array('min_length' => 'A senha contém no mínimo 6 caracteres.'))

};
```

Para realização do login, primeiramente ele faz a validação do que foi enviado pelo formulário, para isso ele usa a biblioteca de validação do CodeIgniter. Eu preciso dizer:

```
$this->form_validation->set_rules('[nome-do-campo]', '[nome-no-label]',  
'[regras-de-validação]')
```

Em seguida, é feito o if para realizar a validação dos dados enviados, caso haja algum problema ele volta para o método index, que mostrará a tela de login novamente. Na tela de login as mensagens de erro serão exibidas.

Caso a validação esteja ok, ele irá para o else. Lá o model de usuario é utilizado para procurar o usuário pelo e-mail e senha.

```
if ($this->form_validation->run() == FALSE) {  
    $this->index();  
} else {  
    $user = $this->Usuario_model->login(["email"=>$_POST["email"],  
        "senha"=>$_POST["senha"]]);
```

Segue o código do método login do model:

```
public function login($data){  
    $data["senha"] = sha1($data["senha"]);  
    return $this->findOne($data);  
}
```

Ele apenas criptografa a senha com sha1, para poder procurá-la no banco, lembre-se que ao cadastrar usuários, devemos criptografá-la para salvar no banco. Evitando que as senhas dos usuários fiquem expostas. Em seguida ele retorna apenas o primeiro registro que encontrar.

Voltando para o controller de login, ele verifica se foi encontrado aquele usuário no banco. Se o \$user for diferente de nulo é porque encontrou, então ele cria as chaves, nome e email na sessão do PHP e redireciona para a página pós login.

```
if ($user != null){  
    $_SESSION["nome"] = $user->nome;  
    $_SESSION["email"] = $user->email;  
  
    redirect("usuarios/index/");  
} else {
```

Caso contrário, ele redireciona para a página de login, com a mensagem de erro. Oflashdata é um tipo de mensagem que será exibida apenas uma vez.

```
$this->session->set_flashdata("message","E-mail ou senha  
incorretos.");
```

```

        redirect("login/index/");
    }
}
}

```

Abra o arquivo views/login.php, observe que estamos enviando o formulário para o método login do controller login.

```

<form class="ui large form" action="<?=site_url()?>/login/login/"
method="POST">

<div class="ui stacked segment">

<div class="field">

<div class="ui left icon input">

<i class="user icon"></i>

<input type="text" name="email" placeholder="Digite seu e-mail">

</div>

<?=form_error('email')?>

</div>

```

Além disso, temos o **form_error**, que irá imprimir a mensagem de erro de validação daquele campo específico, caso a validação esteja ok ele não irá imprimir nada. Todo formulário que precisar de validação deverá conter essa função.

Mais em baixo temos um if, se houver alguma mensagem para ser exibida ele irá imprimi-la. Lembrando que foi utilizado oflashdata, que é um tipo de mensagem que só aparecerá uma vez, se o usuário apenas atualizar a página ela não aparecerá novamente.

```

<?php if ($this->session->flashdata('message')): ?>

<div class="ui red message">

<?=$this->session->flashdata('message');?>

</div>

<?php endif; ?>

```

Uma vez que o usuário esteja dentro do sistema, o menu irá aparecer, e terá a opção **logout**. Quando ele clicar, irá para o método logout do controller login, que irá destruir a sessão, apagando todos os dados que estão nela e redireciona para a tela de login.

```
public function logout(){  
    session_destroy();  
    redirect("login/index/");  
}
```

Em seguida, vem o método cria_usuario, que serve apenas para esse exemplo, apague esse método quando estiver reaproveitando esse projeto para o seu sistema. Para esse método basicamente estamos sempre criando um usuário chamado admin.

```
$data = [ "nome"=>"admin", "email"=>"admin@admin.com", "senha"=>"123456" ];  
if ($this->Usuario_model->save($data)) {
```

Menu

Para entender o menu, abra o arquivo views/layout/header.php. Foi feito um if para verificar se o usuário já está logado. Se estiver, ele mostra o menu. Observe como foi feito o link para a imagem. Lembre-se, .js, .css, .pdf e imagens sempre deverão ficar dentro da pasta **static**.

```
<?php if (isset($_SESSION['email'])): ?>  
  
<div class="ui top fixed menu">  
  
<div class="item">  
  
  
  
</div>
```

```

<a href="<?=site_url()?>/usuarios/" class="item">Usuários</a>
<a href="<?=site_url()?>/setores/" class="item">Setores</a>
<a href="<?=site_url()?>/grupos/" class="item">Grupos</a>
<a href="<?=site_url()?>/login/logout/" class="item"><?=$_SESSION['email']?> |
Logout</a>
</div>

```

Validação do formulário

A validação está sendo feita somente no PHP, o ideal é que ela seja feita também no JavaScript, para evitar requisições desnecessárias. Porém ela sempre deverá existir no lado servidor (no PHP).

Abra o controller de usuarios. No método salvar temos:

```

#a senha so sera validada se for num novo registro
#ou se enviado algo diferente de branco
#pq quando ela nao é enviada, significa que nao é pra alterar
if (val($_POST,"senha") != "" || val($_POST,"id") == ""){
$this->form_validation->set_rules('senha', 'Senha', 'min_length[6]',
    array('min_length' => 'Defina uma senha com no mínimo 6 dígitos.')
);
$this->form_validation->set_rules('senhaConfirm', 'Confirmação da senha',
'required|matches[senha]');
}
$this->form_validation->set_rules('nome', 'Nome', 'required');
$this->form_validation->set_rules('email', 'E-mail', 'required|valid_email');
$this->form_validation->set_rules('setores_id', 'Setor', 'required');

```

O if significa que a validação do campo de confirmação de senha só será realizada se uma senha tiver sido digitada no campo

senha. Isso porque quando um usuário for salvo com a senha em branco a senha dele não será modificada no banco de dados.

Observe que o campo e-mail possui duas validações, `required` e `valid_email`. É possível ter várias outras validações, veja mais exemplos:

https://www.codeigniter.com/user_guide/libraries/form_validation.html

Dentro do método salvar de usuários existe um `if` que não foi mostrado aqui, a explicação desse `if` ficará no detalhamento do upload de arquivos.

Em seguida sempre teremos o `if` para validar:

```
if ($this->form_validation->run() == FALSE) {  
    $this->index();  
} else {  
    {se chegar até aqui está tudo certo com o formulário, continue...}}
```

Observe que no formulário, sempre teremos o **form_error** para exibir as possíveis mensagens de erro. Lembre-se de editar o parâmetro com o nome do input.

<label>Nome

<input type="text" name="nome" value="<?=val(\$dados, 'nome')?>">

<?=form_error('nome')?>

</label>

<label>E-mail

<input type="text" name="email" value="<?=val(\$dados, 'email')?>">

<?=form_error('email')?>

</label>

Dentro do formulário também encontramos a impressão das mensagens de erro. É preciso as 3 linhas, porque podemos ter mensagens de erro (vermelho), sucesso(verde) ou warning(amarelo).

```
<?=$this->session->flashdata('error')?>
<?=$this->session->flashdata('success')?>
<?=$this->session->flashdata('warning')?>
```

O CSS dessas mensagens foi definido no método salvar, logo após ele inserir no banco.

```
if ($obj == ""){
    $this->session->set_flashdata("error","<div class='ui red message'>Falha
ao salvar.</div>");
} else {
    $this->session->set_flashdata("success","<div class='ui green
message'>Salvo com sucesso.</div>");
}
```

O warning só vai ser exibido quando um registro for deletado:

```
public function deletar($id){
    $this->Usuario_model->delete($id);
    $this->session->set_flashdata("warning","<div class='ui yellow
message'>Registro deletado.</div>");
    redirect("usuarios/index");
}
```

Caso você tenha interesse em mudar as mensagens de erro, elas encontram-se em
application/language/portugues/form_validation_lang.php ou
application/language/portugues/upload_lang.php

Upload

Para o exemplo do upload, adicionamos a opção de enviar uma foto no formulário do usuário. Para isso é preciso modificar a declaração do form adicionando o enctype, obrigatoriamente ele deve ser POST:

```
<form class="ui form column stackable grid"
action="<?=site_url()?>/usuarios/salvar" method="post"
enctype="multipart/form-data">
```

Também adicionamos o campo no formulário para enviar a foto:

```
<label>Foto

    <input type="file" name="foto">

    <?=form_error('foto')?>

</label>
```

Como é só um exemplo, a foto será impressa somente abaixo do campo, o if é para imprimir a imagem apenas se existir alguma imagem vinculada.

```
<?php

if (val($dados,"foto") != ""){

    print "<img class='ui tiny circular image'
src='".base_url()."uploads/{$dados['foto']}' />";

} ?>
```

O formulário irá enviar normalmente para o método salvar do controller usuarios. Então o primeiro IF irá verificar se foi enviada alguma foto. O **\$this->upload_name** é o nome do campo no formulário, foi definido no topo da classe.

```
#upload de foto com validacao

if (isset($_FILES[$this->upload_name]) && $_FILES[$this->upload_name]["name"]
!= ""){

    #o arquivo será salvo e feita a validacao
```

```

        $this->form_validation->set_rules($this->upload_name, 'Foto',
        'callback_upload_check');
    }

```

Caso algum arquivo tenha sido enviado ele irá fazer ao mesmo tempo o upload e a validação do arquivo. Por isso ele usa o `form_validation` e define que o método **upload_check** irá checar esse campo.

Logo acima do método salvar temos o **upload_check**, que irá utilizar a própria biblioteca do CodeIgniter de upload. Aqui é definido o local para onde a imagem irá, os tipos de arquivo aceitos e o tamanho máximo. Além disso ele limpa o nome do arquivo, **\$this->cleanString()**, para remover caracteres especiais: acentuação, cê-cedilha e outros caracteres que podem gerar problemas. O upload é feito, e caso esteja tudo ok ele salva o novo nome do arquivo em **\$this->upload_result** e retorna true. Caso ocorra alguma falha, ele usará as próprias mensagens da biblioteca de validação.

```

public function upload_check($v){
    #local onde salvará o arquivo sqlite/uploads/
    $config['upload_path'] = './uploads/';
    $config['allowed_types'] = 'jpg|jpeg|png';
    $config['max_size'] = 1000;
    #limpa os caracteres especiais do nome do arquivo
    $config['file_name'] =
    $this->cleanString($_FILES[$this->upload_name]['name']);

    $this->load->library('upload', $config);
    #faz o upload
    if ( ! $this->upload->do_upload($this->upload_name)){

```

```

        $this->form_validation->set_message('upload_check',
$this->upload->display_errors());

        return false;
    } else {

        #Salva o novo nome do arquivo

        $this->upload_result = $this->upload->data();

        return true;
    }
}
}

```

Quando o formulário fizer a validação, ele terá chamado o **upload_check**, que faz o upload e valida. Após a validação do formulário o arquivo já deve estar na pasta de destino, feito isso, basta salvar o nome do arquivo no registro do usuário:

```

#insere o nome do arquivo que foi feito o upload
#para ser salvo no banco de dados
#nesse momento o arquivo ja se encontra em sqlite/uploads/
$_POST["foto"] = $this->upload_result["file_name"];
$obj = $this->Usuario_model->save();

```

No model de usuarios foi criado o campo foto, esse campo guardará apenas o nome do arquivo. Quando for necessário imprimir a imagem, teremos que completar o caminho:

```

<img src='".base_url()."/uploads/{$dados['foto']}' />

```

Relacionamento Um para Muitos

Para o exemplo do relacionamento de um para muitos utilizamos os Setores de uma empresa, onde um setor poderá ter vários usuários.

Após salvar um setor ou abrir ele para edição, a seguinte tabela com o campo para selecionar o usuário irá aparecer, conforme a figura 8:

Nome

Adicionar pessoa

Usuários no setor		
Nome	E-mail	Remover usuário
joao2	joao@gmail.com	Remover
admin	admin@admin.com	Remover

Figura 8 – Tela de cadastro de setores

Para entender, abra a view de setores. Se existir o campo id na variável **\$dados**, quer dizer que estou editando um setor que já está salvo. Nesse caso, podemos permitir a vinculação de usuários, então a tabela aparecerá.

```
<?php if (val($dados,'id') != ""): ?>
```

```
<div class="field">
```

```
    <label>Adicionar pessoa
```

```
        <select name="pessoa_id">
```

```
            <option value=""></option>
```

```
            <?php
```

```
                foreach($pessoas as $p){
```

```
                    print "<option value='{ $p['id']}'>{$p['nome']}</option>";
```

```
                }
```

```

        ?>
    </select>
</label>
</div>

```

Esse **for** imprime todos os usuários que ainda não estão vinculados com o setor. Esses usuários estão no Array **\$pessoas**. Esse Array foi criado no Controller de **setores**, no método **index**:

```

//Seleciona todas as pessoas que ainda nao estao naquele setor
$this->load->model("Usuario_model");


$pessoas = $this->Usuario_model->findNotInSetor($id);


$this->load->view('setores', ["list"=>$pag["list"],
                                "qtd"=>$pag["qtd"],
                                "page"=>$page,
                                "dados"=>$dados,
                                

"pessoas"=>$pessoas

]);

```

Como essa é uma informação de usuários, foi decidido que o método para buscar os dados do banco ficaria no model usuarios. Nesse método procuramos todos os usuários cuja id do setor seja diferente da id passada via parâmetro ou a id do setor ainda seja nula.

```

public function findNotInSetor($idSetor){
    return R::findAll($this->table,"setores_id <> ? or setores_id is null
",[$idSetor]);
}

```

Logo após, teremos a impressão da tabela com os usuários que já estão cadastrados para o setor.

```

<div class="field">

```

```
<table class="ui celled table">
```

```
    <thead>
```

```
        ...
```

```
    </thead>
```

```
    <tbody>
```

Aqui estamos utilizando o `ownUsuariosList`, para que essa variável exista é preciso observar o Model do setores para entender.

```
<?php
```

```
foreach($dados->ownUsuariosList as $ln){
```

```
    print "<tr>";
```

```
    print "<td>{$ln->nome}</td>";
```

```
    print "<td>{$ln->email}</td>";
```

```
    #para remover o usuario eu preciso passar a id_setor e depois id_usuario
```

```
                                print                                "<td><a  
href='".site_url()."/setores/remover_usuario/{$dados['id']}/{ $ln->id}.'>  
Remover </a></td>";
```

```
    print "</tr>";
```

```
}
```

```
?>
```

```
</tbody>
```

```
</table>
```

```
</div>
```

```
<?php endif; ?>
```

No model do setores temos o seguinte IF, se uma pessoa tiver sido vinculada, recupera ela no banco e adiciona ela à lista `ownUsuariosList` do objeto do setor. Quando você usa essa nomenclatura o RedBean já entende que os dois estão relacionados e já cria a ligação de um para muitos. Nesse caso o Setor poderá ter

vários usuários, então o Setor terá esse Array para ter vários usuários dentro dele. A regra para o nome dessa variável é:

Sempre começar com own + NomeDaTabela + List, obedecendo o camelCase:

```
#salva a pessoa no setor
```

```
if (val($data, 'pessoa_id') != "" ){  
    $usu = R::load("usuarios", $data['pessoa_id']);  
    $obj->ownUsuariosList[] = $usu;  
}
```

A outra forma de fazer esse relacionamento, seria:

```
#adiciona o setor no usuario
```

```
if (val($data, "setores_id") != ""){  
    $set = R::load("setores", $data["setores_id"]);  
    $obj->setores = $set;  
}
```

Essa segunda forma foi utilizada no model de usuários, porque eu posso a partir do formulário de usuários indicar qual é o setor.

As duas formas iria gerar o relacionamento de um para muitos, criando o campo **setores_id** em **usuarios**, como mostra a figura 9:

▼	setores	CREATE TABLE `set
	id	INTEGER
	nome	TEXT
>	sqlite_sequence	CREATE TABLE sqli
▼	usuarios	CREATE TABLE `usi
	id	INTEGER
	nome	TEXT
	email	TEXT
	setores_id	TEXT
	senha	TEXT
	grupos_id	INTEGER
	gruposusuarios_id	INTEGER
	tipo	INTEGER
	foto	TEXT

Figura 9 - Banco de dados

Na view de usuarios temos o select para o setor, que está imprimindo todos os setores.

```
<select name="setores_id">
  <option></option>
<?php
foreach($setores as $s){
    #para ja vir selecionado o setor do usuario, caso tenha sido escolhido
    if ($dados['setores_id'] == $s['id']){
        $selected = "selected";
    } else {
        $selected = "";
    }
    print "<option $selected value='{${s['id']}}'>${s['nome']}</option>";
}
?>
```

```
</select>
```

A variável \$setores foi criada no **index** do controller de usuários:

```
//recupera todos os setores para o select de setor
```

```
$this->load->model("Setor_model");
```

```
$setores = $this->Setor_model->all();
```

```
$this->load->view('usuarios', ["list"=>$pag["list"],  
                                "qtd"=>$pag["qtd"],  
                                "page"=>$page,  
                                "dados"=>$dados,  
                                "setores"=>$setores,  
                                "tiposUsuarios"=>$tiposUsuarios]);
```

Ainda na view de usuarios estamos imprimindo a tabela já com o setor daquele usuário. Para evitar problemas adicionamos o @ antes da linha, isso serve para não imprimir erros nessa linha. Isso porque podem existir usuarios sem setor, ele tentará imprimir o nome de algo nulo, então o @ oculta esse erro.

```
foreach($list as $ln){  
  
    print "<tr>";  
    print "<td><a href='".site_url()."/usuarios/index/{ $ln->id}'> Editar  
</a></td>";  
    print "<td>{ $ln->nome}</td>";  
    print "<td>{ $ln->email}</td>";  
    @print "<td>{ $ln->setores->nome}</td>";  
    print "<td><a  
onclick='confirmDelete(\"".site_url()."/usuarios/deletar/{ $ln->id}\")'>  
Deletar </a></td>";
```

```
print "</tr>";  
}
```

Relacionamento Muitos para Muitos

Para o relacionamento de muitos para muitos utilizamos os grupos, vários usuários podem estar em um grupo e um grupo pode conter vários usuários.

A mesma lógica do formulário anterior é seguida, a tabela para vinculação só irá aparecer após o grupo ter sido salvo. No model de grupos nós temos o IF para verificar se um usuário foi vinculado. Em seguida buscamos ele no banco e criamos uma tabela para realizar o relacionamento entre a tabela de usuarios e grupos, infelizmente o nome dessa tabela só pode conter letras, ou seja, não pode ser grupos_usuarios nem grupos-usuarios, também não podem existir letras maiúsculas no nome das tabelas. Após criar o objeto de gruposusuarios, inserimos o usuario e o grupo na tabela gruposusuarios e salvamos.

```
#salva a pessoa no grupo  
if (val($data, 'pessoa_id') != "" ){  
    $usu = R::load("usuarios", $data['pessoa_id']);  
    #nomes de tabelas nao podem ter _ - ou letras maiusculas  
    $assoc = R::dispense("gruposusuarios");  
    #o nome das propriedades tem que ser igual ao nome das tabelas (plural)  
    $assoc->usuarios = $usu;  
    $assoc->grupos = $obj;  
  
    R::Store($assoc);  
}
```

Para imprimir os usuarios do grupo fazemos algo parecido com o que foi feito em setores, usamos o `ownGruposusuariosList`, que irá trazer todas as ligações desse grupo, e imprimimos o usuário que está em cada uma dessas ligações:

```
foreach($dados->ownGruposusuariosList as $grouplist){
    $user = $grouplist->usuarios;
    print "<tr>";
    print "<td>{$user->nome}</td>";
    print "<td>{$user->email}</td>";
    #para remover o usuario eu preciso passar a id_setor
    #depois a id do registro da tabela gruposusuarios
    print "<td><a
href='".site_url()."/grupos/remover_usuario/{$dados['id']}/{$grouplist->id}'>
Remover </a></td>";
    print "</tr>";
}
```

O único detalhe é que para desfazer o vínculo, estamos passando a id do setor, para que ele retorne para a mesma página e a id do vínculo, ou seja, a id da linha na tabela `gruposusuarios`, veja o método no controller de Grupos:

```
public function remover_usuario($grupo_id, $assoc_id){
    $this->Grupo_model->remove_usuario($assoc_id);
    redirect("grupos/index/" . $grupo_id );
}
```

Veja no model de grupos:

```
public function remove_usuario($id_assoc){
    $obj = R::load("gruposusuarios", $id_assoc);
    R::Trash($obj);
}
```

```
}
```

Confirmação antes de deletar um registro

Você deve ter notado que antes de deletar um registro, sempre há uma janela para confirmar sua exclusão. Isso é importante para evitar exclusões acidentais.

Veja que em todos os botões de deletar estamos chamando a função `confirmDelete` do JavaScript:

```
foreach($list as $ln){  
    print "<tr>";  
    print "<td><a href='".site_url()."/grupos/index/{$ln->id}'> Editar  
</a></td>";  
    print "<td>{$ln->nome}</td>";  
    print "<td><a  
onclick='confirmDelete(\"".site_url()."/grupos/deletar/{$ln->id}\" )'> Deletar  
</a></td>";  
    print "</tr>";  
}
```

Como parâmetro passamos a url que irá remover aquele registro. A função em JavaScript encontra-se em: `sqlite_completo/static/delete.js`

```
function confirmDelete(url){  
    $('.ui.basic.modal').modal('show');  
    $('#confirmDeleteBtn').click(function(){  
        window.location = url;  
    })  
}
```

Essa função, chama o modal do Semantic CSS, esse modal está na View `layout/bottom.php`, o modal é genérico e funcionará para

qualquer registro. Quando o botão `confirmDeleteBtn` é clicado, a função encaminha para a URL que foi passada via parâmetro.

```
<div class="ui basic mini modal">
<div class="ui icon header">
    <i class="trash icon"></i>Confirmação</div>
<div class="content center">
    <p>Você realmente deseja deletar esse registro?</p>
</div>
<div class="actions">
<div class="ui red basic cancel inverted button">
    <i class="remove icon"></i>No
</div>
<div id="confirmDeleteBtn" class="ui green ok inverted button">
    <i class="checkmark icon"></i>Yes
</div>
</div>
</div>
```

Filtros/Buscas

Para criar o campo de busca é preciso ter outro formulário, esse formulário foi inserido acima da tabela onde será aplicado o filtro:

```
<form class="ui form column stackable grid" action="<?==site_url()?>/usuarios"
method="GET">
    <div class="fields">
        <input name="busca" placeholder="Pesquisar."
value="<?==val($_GET, 'busca')?>"/>
        <button class="ui blue button" type="submit">Pesquisar</button>
    </div>
</form>
```

Observe que esse form é via GET, isso é importante. Isso porque o usuário pode querer copiar o link da busca e enviar para alguém. Perceba também que é um único campo que irá buscar em todos os campos. Esse formulário não possui validação e está sendo enviado para o **index** do controller usuarios. Perceba que não foi especificado usuarios/index, mas como já foi dito, se nada for especificado o padrão é ir para o **index**.

```
//busca todos os registros para a listagem
```

```
$pag = $this->Usuario_model->pagination($page, val($_GET, "busca"));
```

A linha acima indica ao model o que será pesquisado, ou seja, o valor que foi digitado no campo busca que veio via GET.

O método **pagination** do **AbstractModel** basicamente irá fazer um where com todos os campos que forem definidos na variável \$searchFields, se essa variável não for definida ele irá usar a variável \$fields.

```
class Usuario_model extends AbstractModel {  
    public $table = "usuarios";  
    public $fields = ["nome", "email", "senha", "tipo", "foto"];  
    public $tiposUsuarios = [1=>"Professor", 2=>"Técnico", 3=>"Bolsista"];  
    public $searchFields = ["nome", "email"];
```

Veja que essa variável só foi definida no Model de usuários porque eu não quero que ele pesquise nos campos senha, tipo e foto.

Caso você precise de algo mais específico, você sempre poderá sobrescrever o método do **AbstractModel** e criar o seu próprio SQL para buscar nas tabelas.

Removendo o index.php da URL

Para remover o index.php da URL, crie um arquivo chamado .htaccess no root do seu site (O root é a pasta principal, a que contém as pastas application, static, system...). Perceba que não há extensão no nome do arquivo, é apenas .htaccess, caso você não consiga criar um arquivo assim, copie e cole o arquivo que tem esse nome de dentro da pasta **application**.

O conteúdo desse novo arquivo deve ser:

```
RewriteEngine on
RewriteCond $1 !^(index\.php|resources|robots\.txt)
RewriteCond %{REQUEST_FILENAME} !-f
RewriteCond %{REQUEST_FILENAME} !-d
RewriteRule ^(.*)$ index.php/$1 [L,QSA]
```

Além disso é preciso configurar o application/config/config.php com:

```
$config["index_page"] = "";
```

Feito isso suas páginas poderão ser acessadas com:

<http://localhost/sqlite/usuarios/index>

[http://\[endereco\]/\[pastaprincipal\]/\[controller\]/\[metodo\]](http://[endereco]/[pastaprincipal]/[controller]/[metodo])

No lugar de:

<http://localhost/sqlite/index.php/usuarios/index>

[http://\[endereco\]/\[pastaprincipal\]/index.php/\[controller\]/\[metodo\]](http://[endereco]/[pastaprincipal]/index.php/[controller]/[metodo])

Debugger

Você deve ter percebido que esse projeto tem uma área que mostra todos os SQL, figura 10, que foram executados no banco, isso te ajudará a encontrar problemas no seu software.

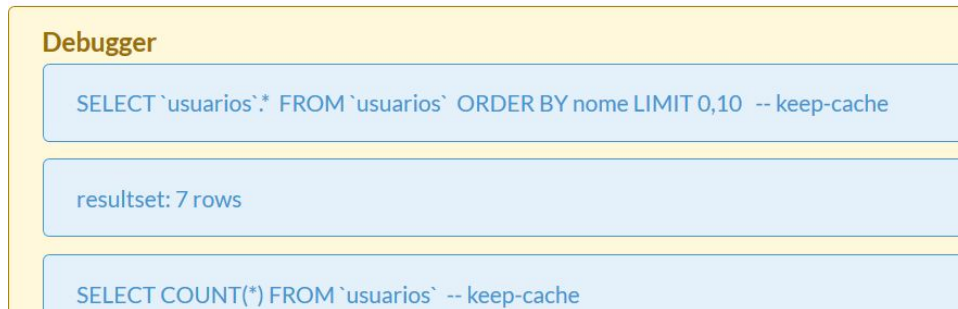


Figura 10 - Debugger

Quando você não quiser mais ver essas mensagens, abra o arquivo `sqlite_completo/application/index.php` e mude a linha

```
define('ENVIRONMENT', isset($_SERVER['CI_ENV']) ? $_SERVER['CI_ENV'] :  
'development');
```

Para:

```
define('ENVIRONMENT', isset($_SERVER['CI_ENV']) ? $_SERVER['CI_ENV'] :  
'production');
```

Caso precise ver novamente, mude para development.